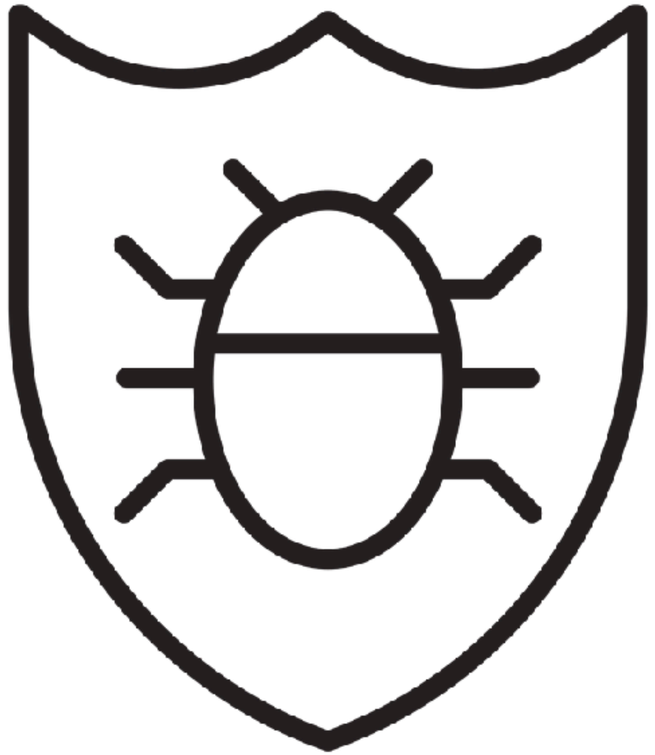
Software Verification

Fundamentals of Software Testing



Bin Lin

Coming Lectures…

Week 3

- Fundamentals of Software Testing
- Specification-Based Testing
- Test-Driven Development

Week 4

- Structural Testing
- Mutation Testing

Week 5

- Test Doubles
- Writing Larger Tests

Software Systems Often Fail

Therac-25

a computer-controlled radiation therapy machine

patients were given massive overdoses of radiation

3 patients were killed

concurrent programming errors!

Image: https://hci.cs.siue.edu/NSF/Files/Semester/Week13-2/PPT-Text/Slide13.html

Software Systems Often Fail

Ariane 5 rocket

maiden flight failed

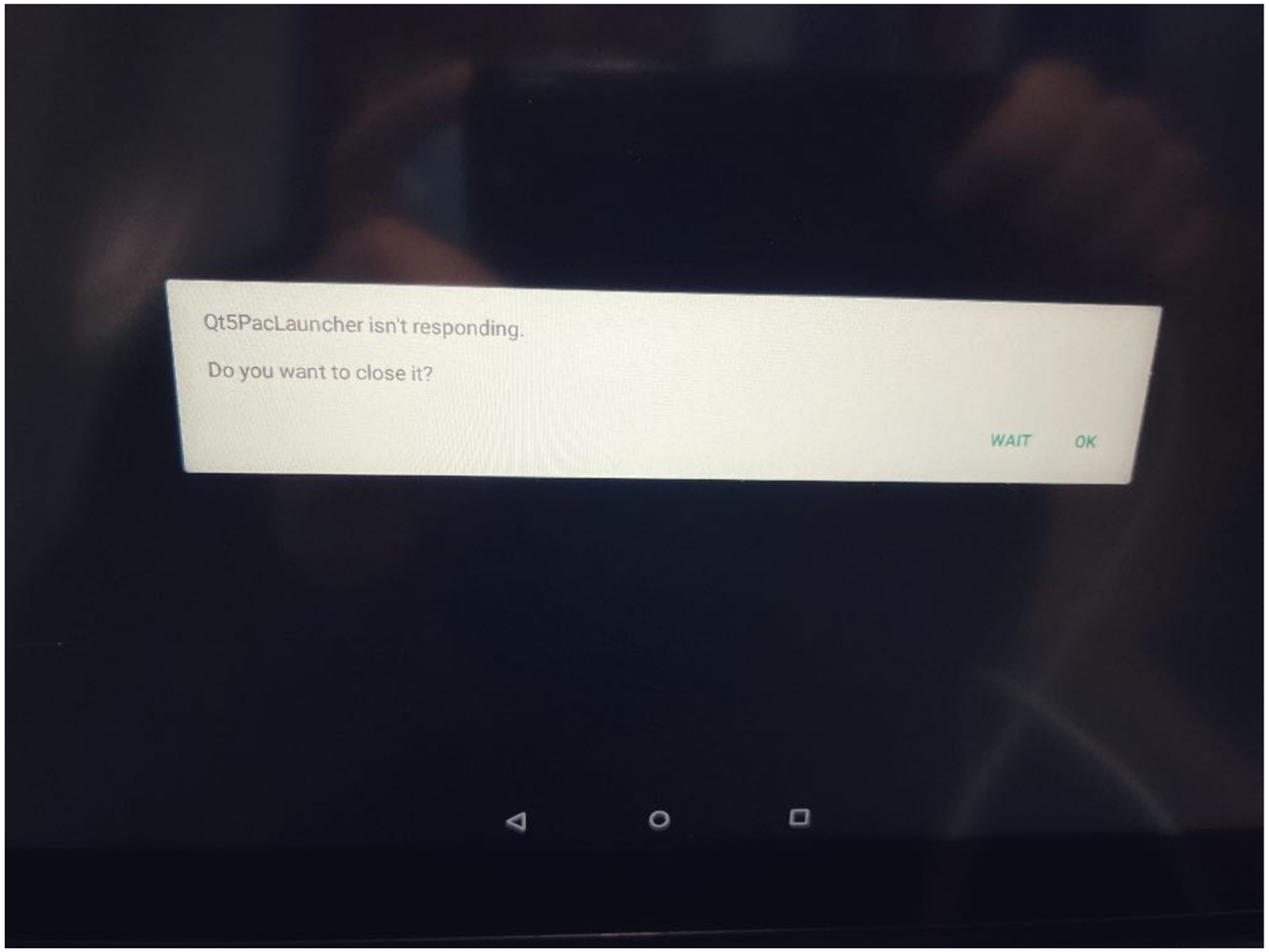
launched on 4 June 1996

one of the most expensive software failures: US$370M

incorrectly reused code from Ariane 4

Image: https://www.esa.int/ESA_Multimedia/Images/2009/09/Explosion_of_first_Ariane_5_flight_June_4_1996

Software Systems Often Fail



Inflight entertainment system

Software Systems Often Fail

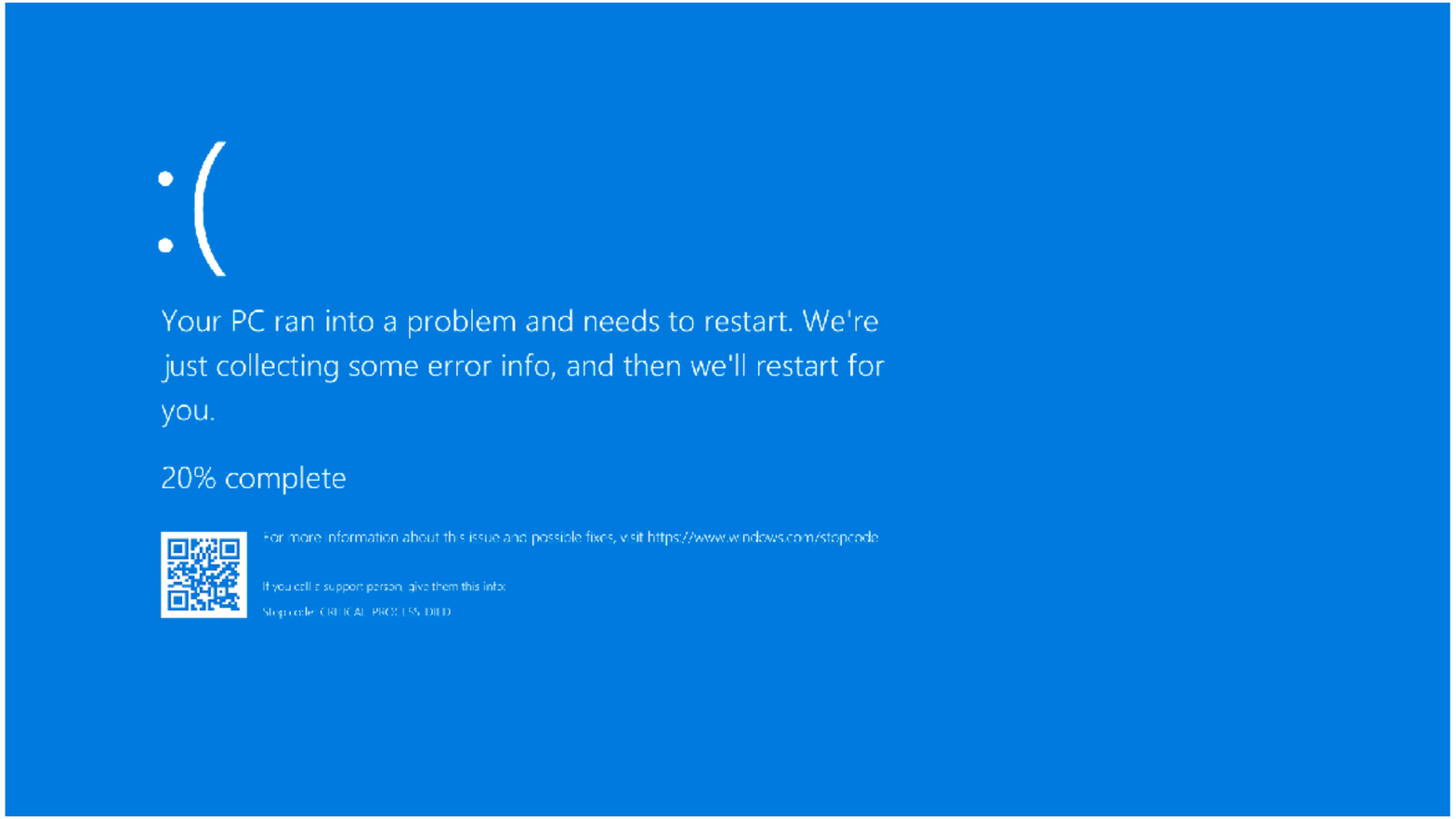


Image: https://en.wikipedia.org/wiki/Blue_screen_of_death

Software Systems Often Fail

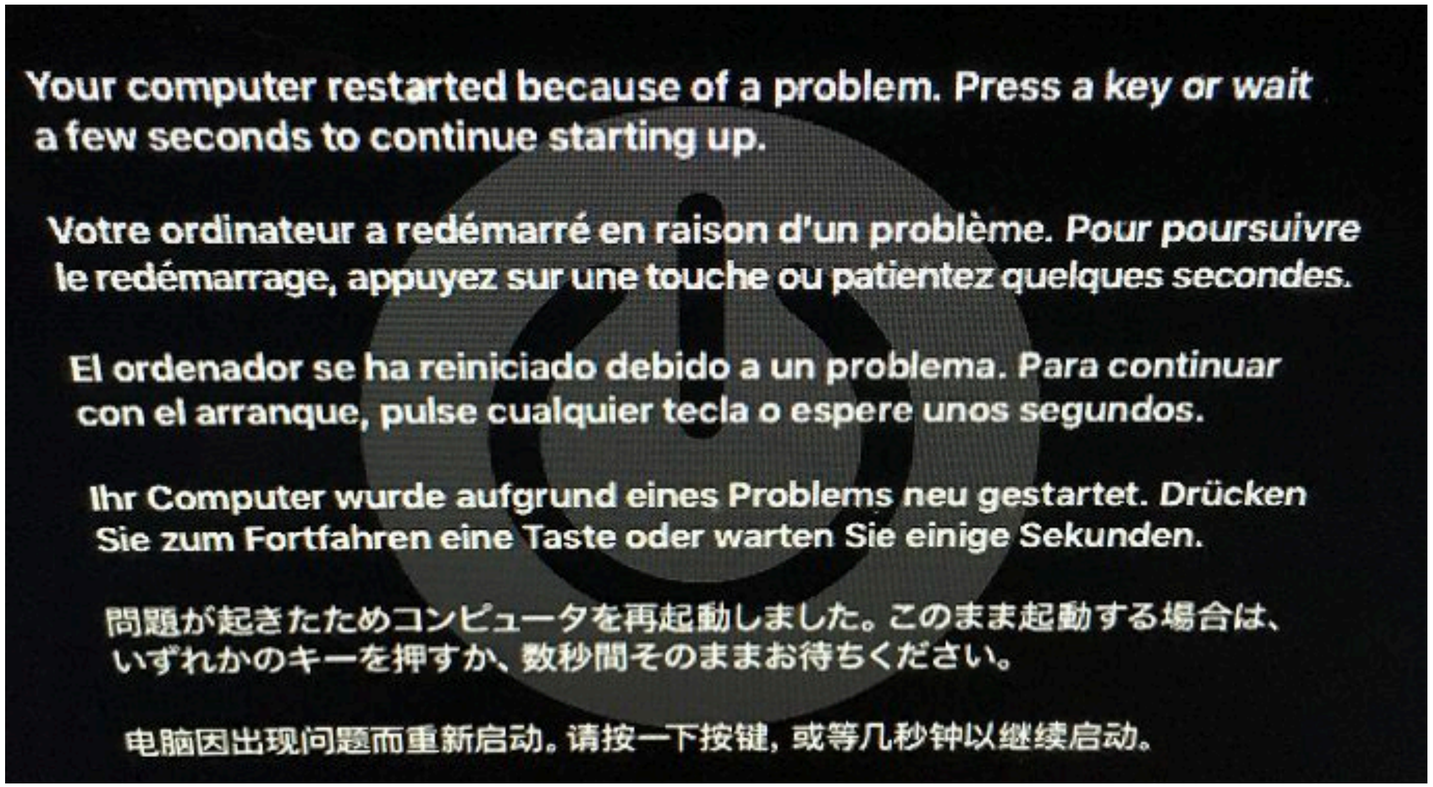


Image: https://appletoolbox.com/your-computer-restarted-because-of-a-problem-mac/

Error, Fault, Failure

Error — people make errors. A good synonym is **mistake**.

human action producing fault

Fault — a fault is the result/representation of an error. **Defect** is a good synonym for fault

missing / incorrect code

Failure — a failure occurs when the code corresponding to a fault executes.

inability of a system to perform required function

Example

```
public static int numZero (int [ ] arr) {
    int count = 0;
    for (int i = 1; i < arr.length; i++){
        if (arr[i] == 0){
            count++;
        }
    }
    return count;
}
```

Example

```
public static int numZero (int [ ] arr) {
    int count = 0;
    for (int i = 1; i < arr.length; i++){
        if (arr[i] == 0){
            count++;
        }
    }
    return count;
}
```

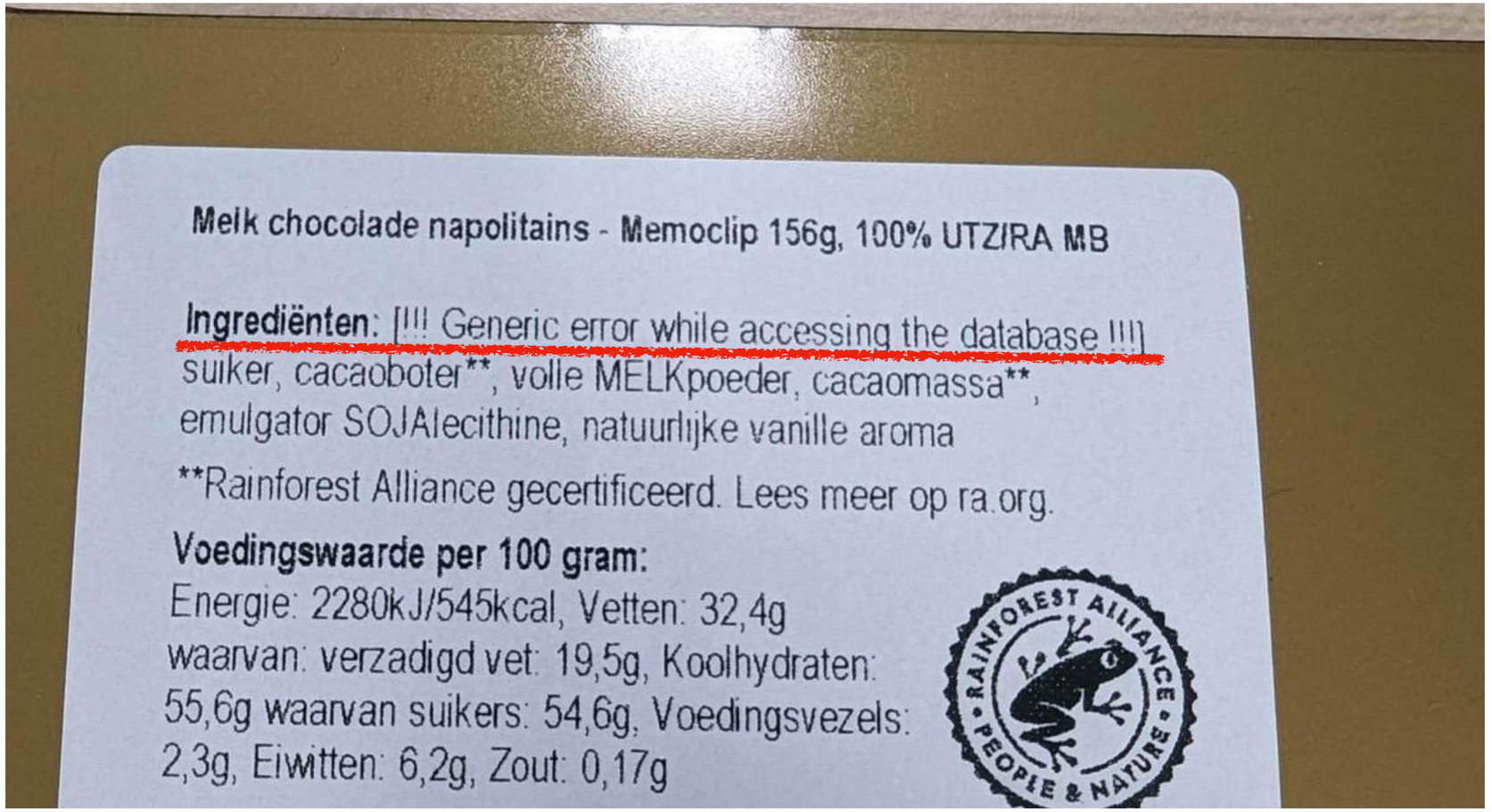
Error: I typed the wrong number in my tiny keyboard

Fault: i should start at 0, not 1

Failure: an input of [0, 2, 7] will lead to an output of 0

Error, Fault, Failure

Ingredient Information: Fault or Failure?



Ingredient information **failure** to deliver correct information, caused by **fault** in code

Bug

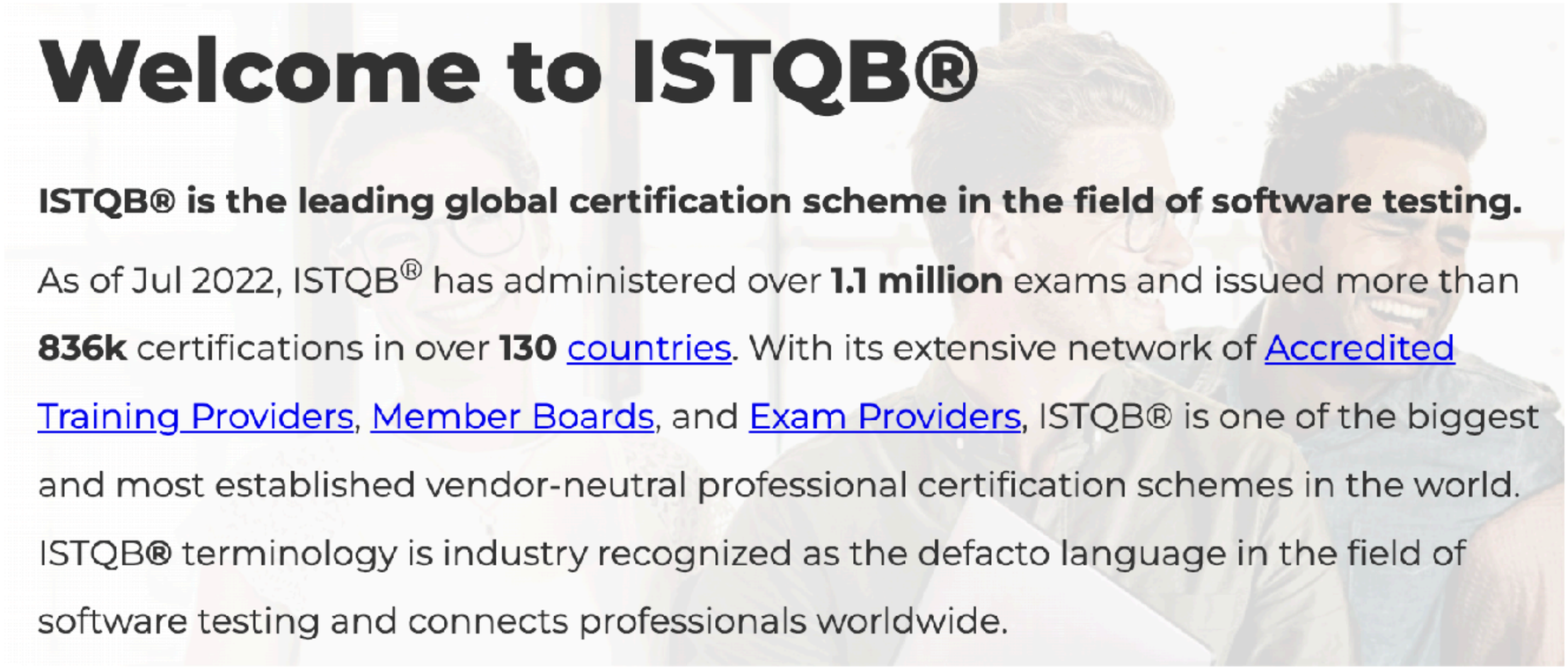
Bug is used **informally.**

Sometimes speakers mean *fault*, sometimes *error*, sometimes *failure*…
often the speaker doesn't know what it means!

A tester's goal is to eliminate faults as early as possible!

Principles of Software Testing

International Software Testing Qualifications Board (ISTQB)



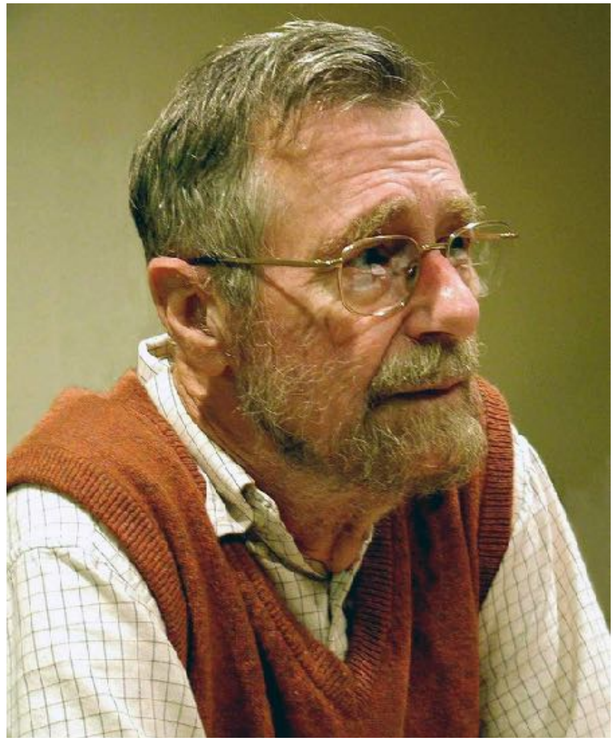
Principles of Software Testing

#1 Testing shows the presence of defects, not their absence.

"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."

Edsger W. Dijkstra

The Humble Programmer,
ACM Turing Lecture 1972



Principles of Software Testing

#2 Exhaustive testing is impossible.

a software system with “only” 300 different flags which can be can be set to true or false

$2^{300} \approx 2*10^{90}$ combinations **>** $\approx 10^{80}$ # atoms in the observable universe

Principles of Software Testing

#3 Early testing saves time and money.

Table 1-5. Relative Costs to Repair Defects when Found at Different Stages of the Life-Cycle

Life Cycle Stage	Baziuk (1995) Study Costs to Repair when Found	Boehm (1976) Study Costs to Repair when Found[a]
Requirements	1X[b]	0.2Y
Design		0.5Y
Coding		1.2Y
Unit Testing		
Integration Testing		
System Testing	90X	5Y
Installation Testing	90X-440X	15Y
Acceptance Testing	440X	
Operation and Maintenance	470X-880X[c]	

[a]Assuming cost of repair during requirements is approximately equivalent to cost of repair during analysis in the Boehm (1976) study.

[b]Assuming cost to repair during requirements is approximately equivalent to cost of an HW line card return in Baziuk (1995) study.

[c]Possibly as high as 2,900X if an engineering change order is required.

"The economic impacts of inadequate infrastructure for software testing." *National Institute of Standards and Technology* 1 (2002).

Principles of Software Testing

#4 Defects cluster together.

Schröter et al. studied bugs in the Eclipse projects
71% of files that imported compiler packages had to be fixed later

Schröter, Adrian, Thomas Zimmermann, and Andreas Zeller. 2006. "*Predicting Component Failures at Design Time*."
In Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering. ACM.

Principles of Software Testing

#5 Beware of the pesticide paradox.

Pesticide paradox: every method you use to prevent or find defects leaves a residue of subtler defects against which those methods are ineffectual.

Re-running the same test suite again and again on a changing program gives a false sense of security: we need variation in testing!

Principles of Software Testing

#6 Testing is context dependent.

Principles of Software Testing

#7 Absence-of-errors is a fallacy.

A software system that works flawlessly but is of no use to its users is not a good software system.

“Building the software right versus building the right software.”

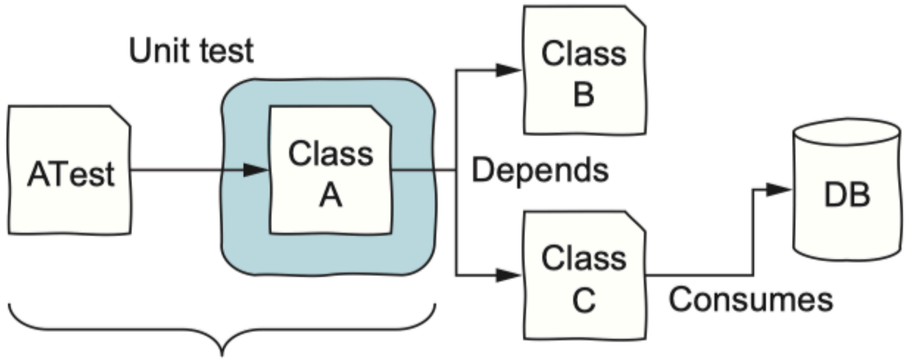
Test Level

Unit testing

Units in isolation

- Unit tests are fast.
- Unit tests are easy to control.
- Unit tests are easy to write.

- Unit tests lack reality.
- Some types of bugs are not caught.

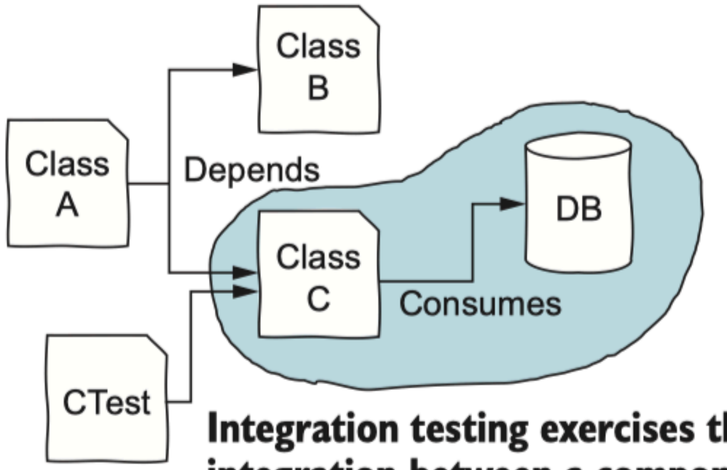


When unit testing class A, our focus is on testing A, as isolated as possible from the rest! If A depends on other classes, we have to decide whether to simulate them or to make our unit test a bit bigger.

Test Level

Integration testing

Interaction between units



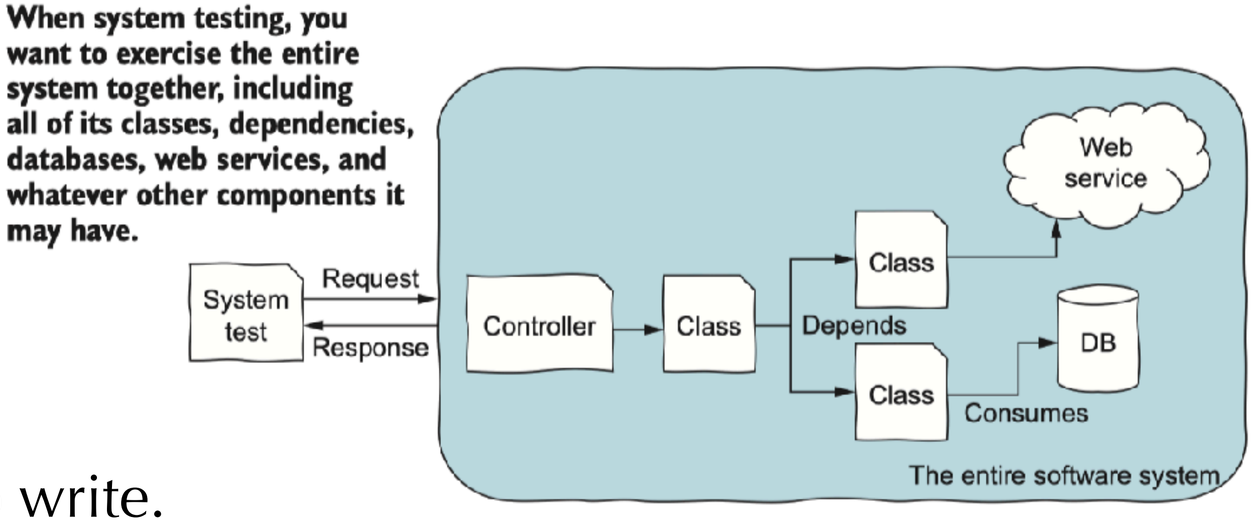
Integration testing exercises the integration between a component of your system and some external component (e.g., a database).

Test Level

System testing

System-level properties

- The obvious advantage of system testing is how realistic the tests are.



- System tests are often slow compared to unit tests.
- System tests are also harder to write.
- System tests are more prone to flakiness.

Test Level

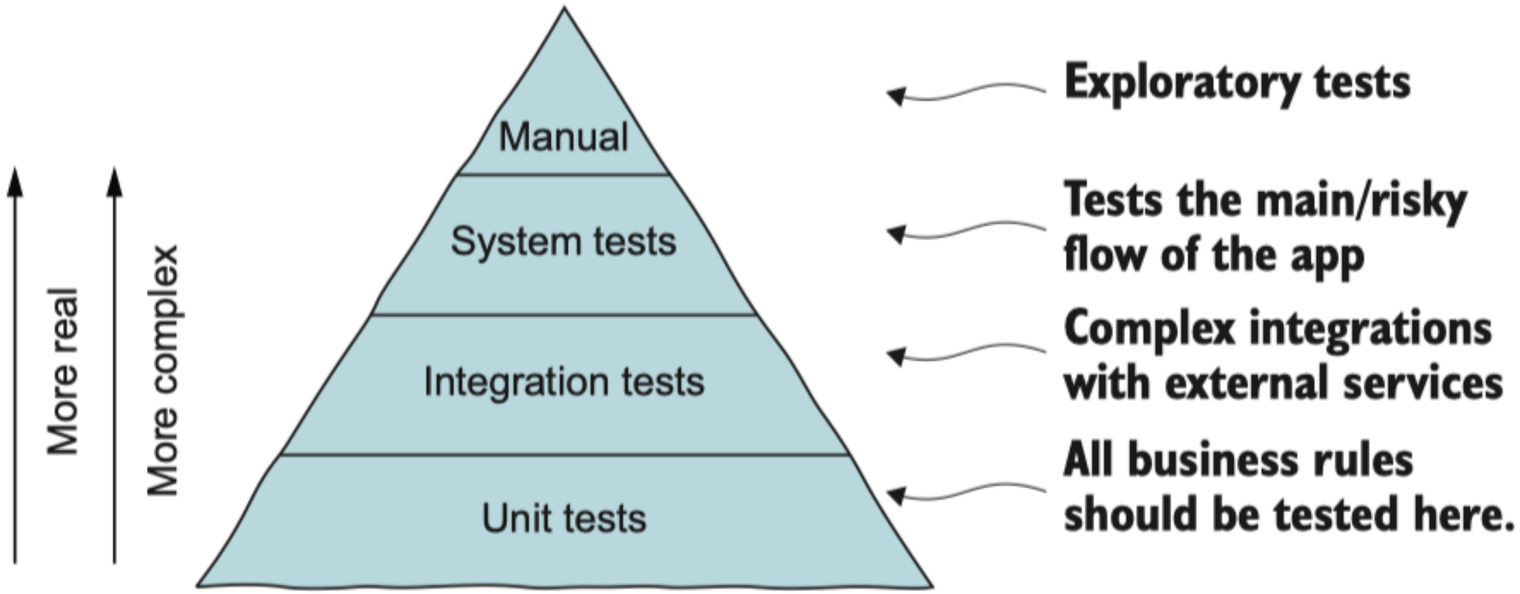
Acceptance testing

Focus on user needs

after System Testing and before delivering the software

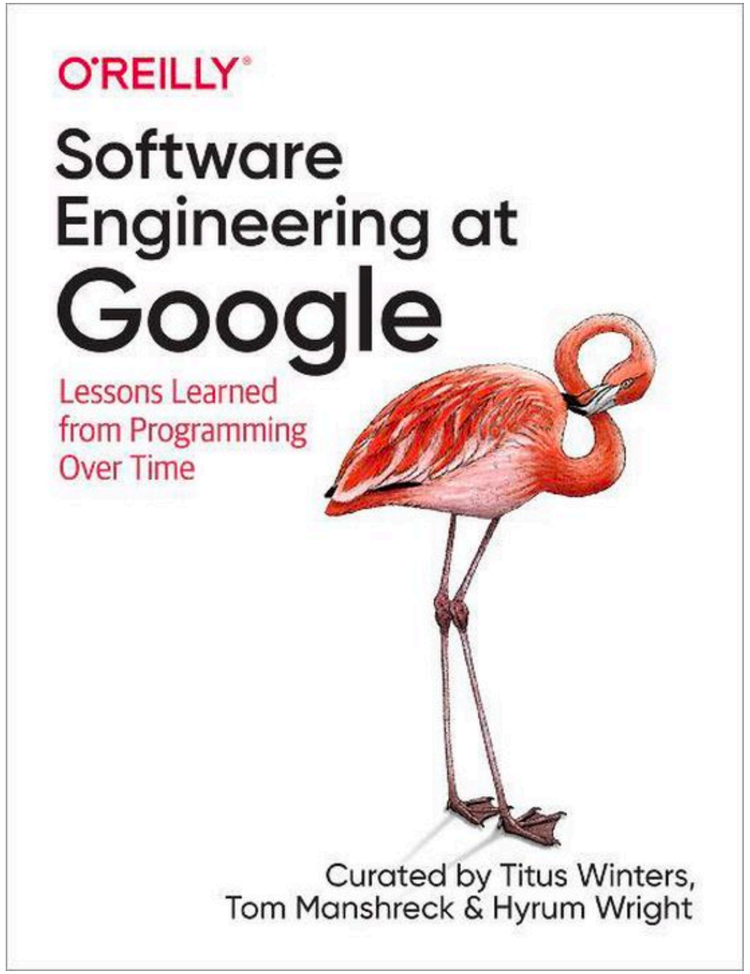
“Formal testing with respect to user needs, requirements, and business processes conducted to determine whether a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether to accept the system.” - ISTQB

The Testing Pyramid

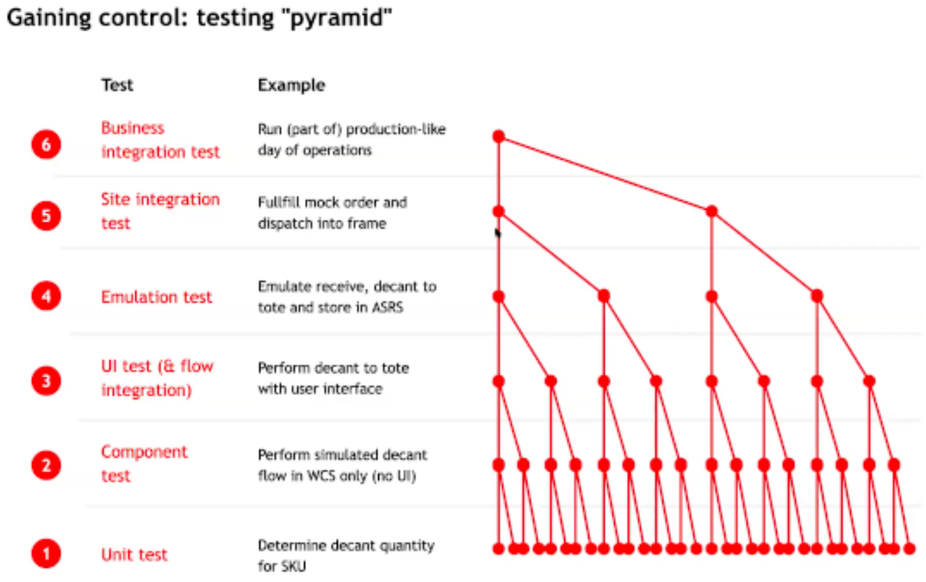


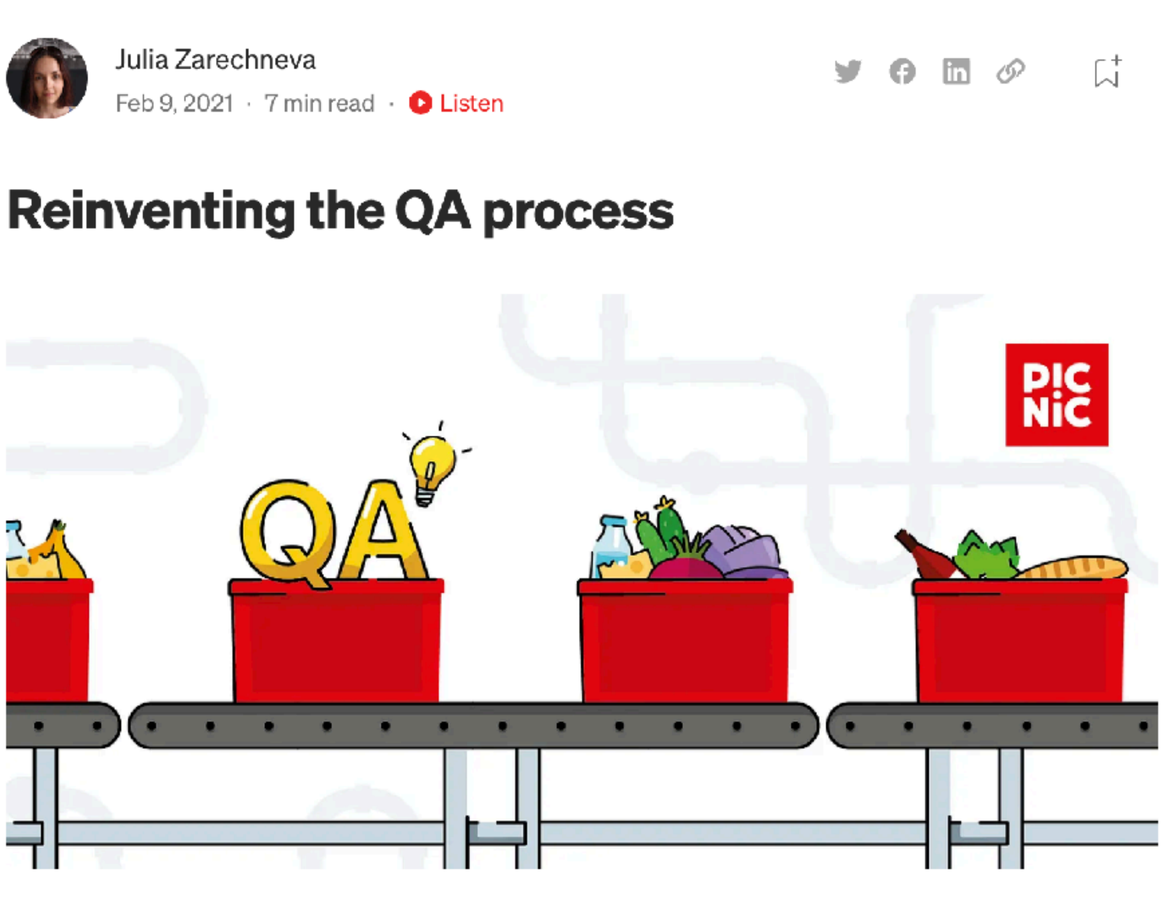
Different Opinions

Google often opts for unit tests, as they tend to be cheaper and execute more quickly. Integration and system tests also happen, but to a lesser extent. According to the authors, around 80% of their tests are unit tests.



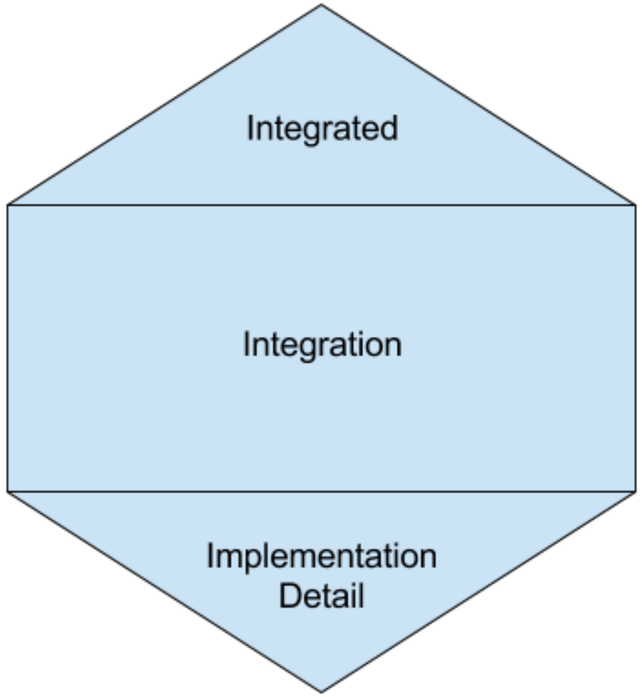
Different Opinions





https://blog.picnic.nl/reinventing-the-qa-process-25854fee51f3

Different Opinions



Microservices Testing Honeycomb

https://engineering.atspotify.com/2018/01/testing-of-microservices/

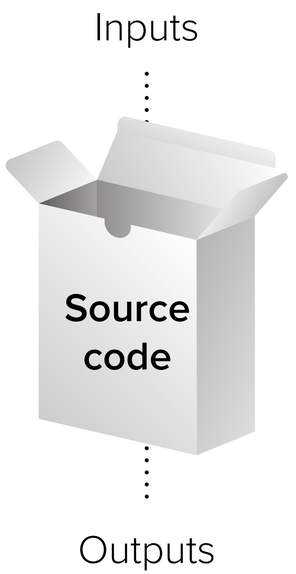
Specification-Based and Structural Testing

Specification-Based Testing (Functional Testing/Black-box Testing)

examines functionality without any knowledge of internal implementation

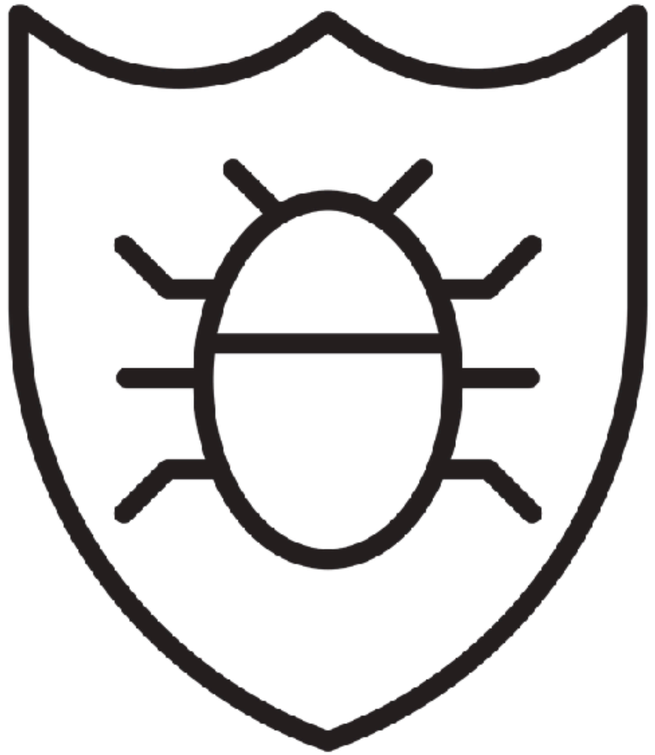
Structural Testing (White-Box Testing)

tests the system based on the knowledge of the program structure



Software Verification

Specification-Based Testing



Bin Lin

Example

Regulation EU261

non-internal EU flights

€250 in respect of flights of 1,500km or less; or
€400 in respect of flights between 1,500km and 3,500km; or
€600 in respect of more than 3,500 km.

int getCompensation(int distance)

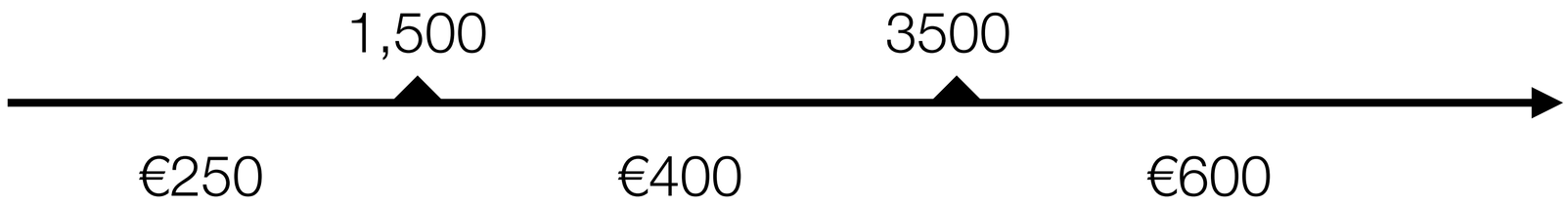
Example

Regulation EU261

non-internal EU flights

€250 in respect of flights of 1,500km or less; or
€400 in respect of flights between 1,500km and 3,500km; or
€600 in respect of more than 3,500 km.

int getCompensation(int distance)



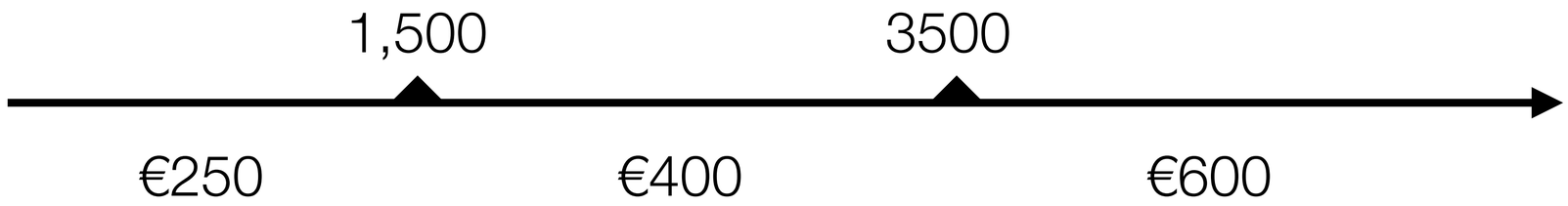
Example

Regulation EU261

non-internal EU flights

€250 in respect of flights of 1,500km or less; or
€400 in respect of flights between 1,500km and 3,500km; or
€600 in respect of more than 3,500 km.

int getCompensation(int distance)



Example

Regulation EU261

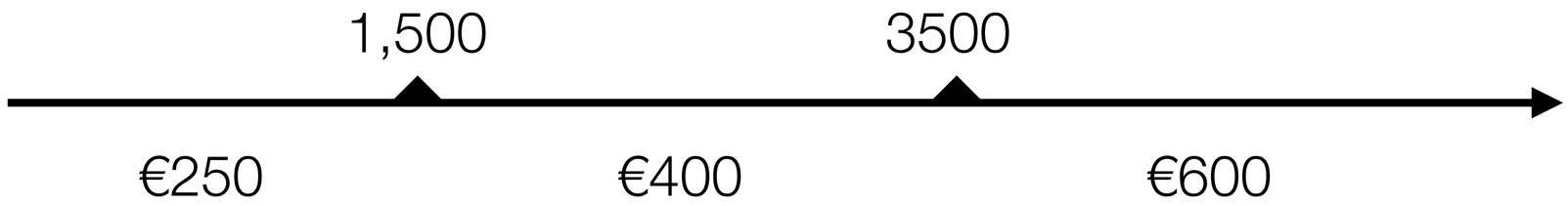
non-internal EU flights

€250 in respect of flights of 1,500km or less; or
€400 in respect of flights between 1,500km and 3,500km; or
€600 in respect of more than 3,500 km.

int getCompensation(int distance)

T1 = 1000
T2 = 2000
T3 = 4000

T1 = 2000
T2 = 2500
T3 = 3000

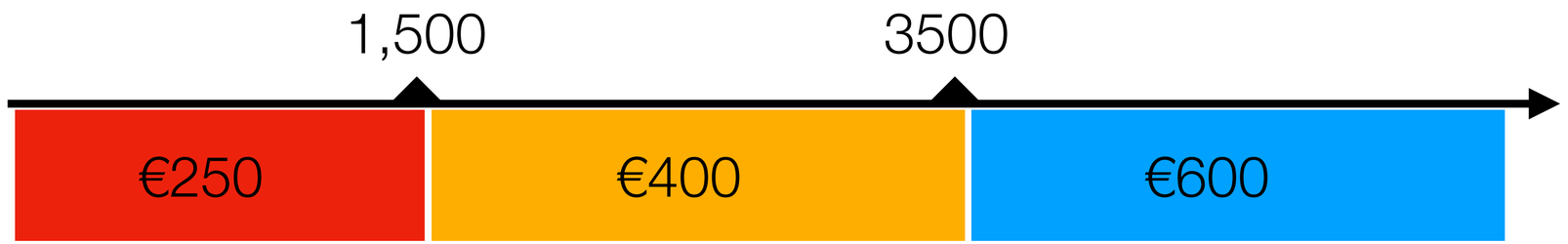


Partitions

Partitions are representative classes of our program

Equivalent partitions: "*A portion of an input or output domain for which the behavior of a component or system is assumed to be the same, based on the specification*" (ISTQB definition)

We should try to reduce the human cost:
having lots of (repeated) tests increase the cost!



Equivalent Partition

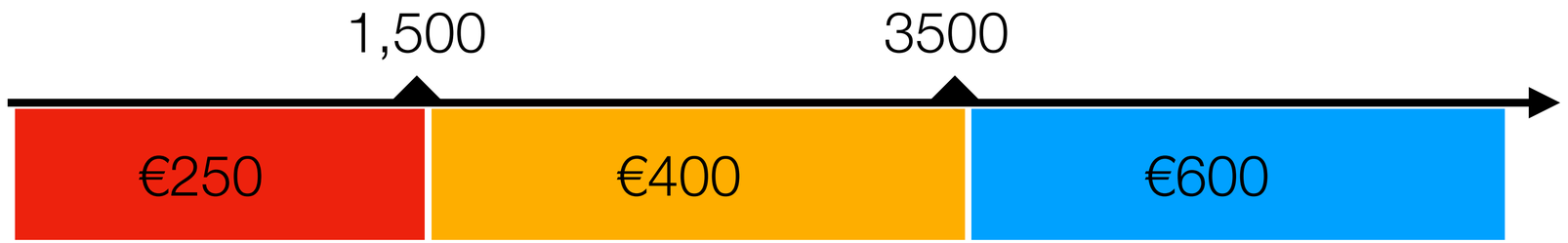
non-internal EU flights

```
if (distance < 1500)
    return 250;
```

€250 in respect of flights of 1,500km or less; or
€400 in respect of flights between 1,500km and 3,500km; or
€600 in respect of more than 3,500 km.

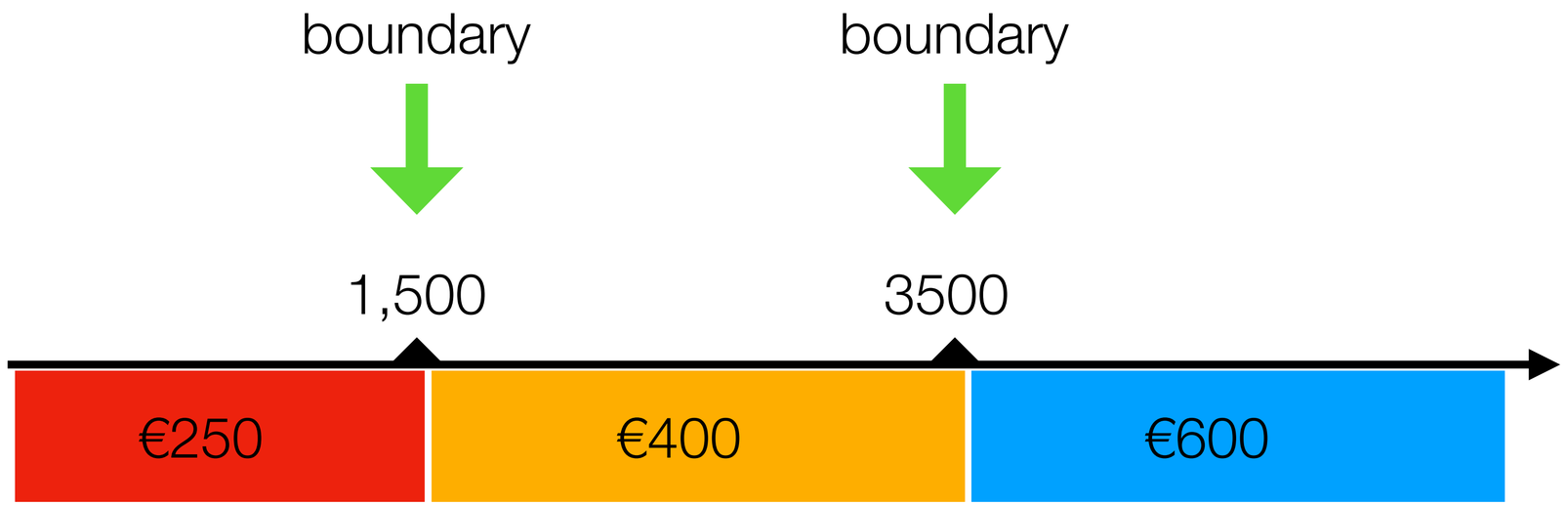
int getCompensation(int distance)

T1 = 1000
T2 = 2000
T3 = 4000



Equivalent Partition

Boundary Analysis!



Boundary Analysis

On point: Exactly on boundary

In point: Makes the condition true

Out point: Makes the condition false

Off point: Flips the outcome for on point and is as close to boundary as possible

Boundary Analysis

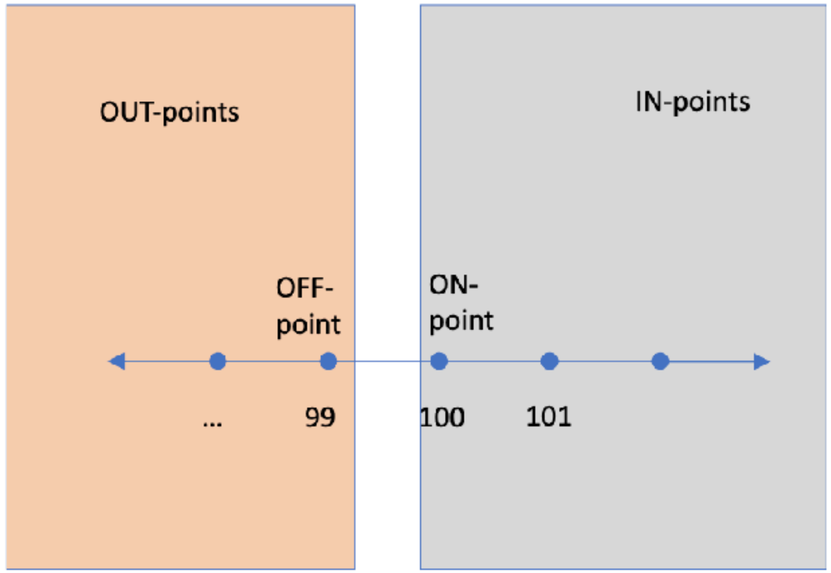
On point: Exactly on boundary

In point: Makes the condition true

Out point: Makes the condition false

Off point: Flips the outcome for on point and is as close to boundary as possible

```
X >= 100
```



Boundary Analysis

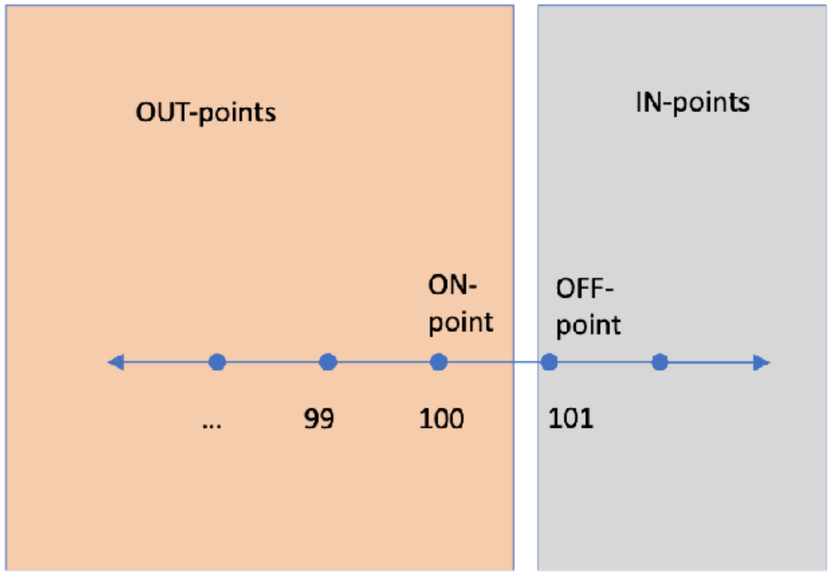
On point: Exactly on boundary

In point: Makes the condition true

Out point: Makes the condition false

Off point: Flips the outcome for on point and is as close to boundary as possible

```
X > 100
```



Boundary Analysis

On point: Exactly on boundary

In point: Makes the condition true

Out point: Makes the condition false

Off point: Flips the outcome for on point and is as close to boundary as possible

On and off points are enough, but feel free to add in and out points!

Boundary Analysis

On point: Exactly on boundary

In point: Makes the condition true

Out point: Makes the condition false

Off point: Flips the outcome for on point and is as close to boundary as possible

```
X == 100
```

What are the on and off points?

On point = 100 (always the value in the condition)
Off point = 101 and 99 (There are two off-points!)

Boundary Analysis

On point: Exactly on boundary

In point: Makes the condition true

Out point: Makes the condition false

Off point: Flips the outcome for on point and is as close to boundary as possible

```
X <= 2.0
```

What are the on and off points?

On point = 2.0
Off point = ? (It depends!)

Specification-Based Testing in a Nutshell

```
public static String repeat(final String str, final int repeat)
```

Repeat a String a specified number of times to form a new String.

@param str: the String to repeat, may be null
@param repeat: number of times to repeat str, negative treated as zero
@return: a new String consisting of the original String repeated, null if null String input

This method was taken from the Apache Commons Lang.

Specification-Based Testing in a Nutshell

```
public static String repeat(final String str, final int repeat)
```

Step 1: Understanding the ***requirements, inputs, and outputs***

The goal of this method is to repeat a String a specified number of times to form a new String.

The program receives two ***parameters***:
- str: the String to repeat
- repeat: number of times to repeat

The program ***returns*** an String of the original String repeated

Specification-Based Testing in a Nutshell

```
public static String repeat(final String str, final int repeat)
```

Step 2: Explore what the program does for various inputs

Examples

Inputs:	***Inputs***:
str: “ab”	str: “a”
repeat: 3	repeat: 2
Output:	***Output:***
“ababab”	“aa”

Specification-Based Testing in a Nutshell

```
public static String repeat(final String str, final int repeat)
```

Step 2: Explore what the program does for various inputs

Examples

Inputs:	***Inputs***:
str: “ab”	str: “a”
repeat: 3	repeat: 2
Output:	***Output:***
“ababab”	“aa”

Specification-Based Testing in a Nutshell

```
public static String repeat(final String str, final int repeat)
```

Step 3: Explore possible inputs and outputs, and identify partitions

1. Each input individually: “What are the possible classes of inputs I can provide?”

2. Each input in combination with other inputs: “What combinations can I try between the str and repeat?”

3. The different classes of output expected from this program: “Does it return a String? Can it return an empty string? Can it return null values?”

Specification-Based Testing in a Nutshell

```
public static String repeat(final String str, final int repeat)
```

Step 3: Explore possible inputs and outputs, and identify partitions

str ***repeat***

Repeat a String a specified number of times to form a new String.

@param str: the String to repeat, may be null
@param repeat: number of times to repeat str, negative treated as zero
@return: a new String consisting of the original String repeated, null if null String input

Specification-Based Testing in a Nutshell

```
public static String repeat(final String str, final int repeat)
```

Step 3: Explore possible inputs and outputs, and identify partitions

str
- Null string
- Empty string
- String of length 1
- String of length 2
- String of length > 2

repeat
- Negative
- 0
- 1
- 2
- > 2

Specification-Based Testing in a Nutshell

```
public static String repeat(final String str, final int repeat)
```

Step 3: Explore possible inputs and outputs, and identify partitions

str
- Null string
- Empty string
- String of length 1
- String of length 2
- String of length > 2

repeat
- Negative
- 0
- 1
- 2
- > 2

"What combinations can I try between the str and repeat?"

Think about an example:
2 inputs: listA, index i
Output: listA[i]
What's the difference here?

Specification-Based Testing in a Nutshell

```
public static String repeat(final String str, final int repeat)
```

Step 3: Explore possible inputs and outputs, and identify partitions

str
- Null string
- Empty string
- String of length 1
- String of length 2
- String of length > 2

repeat
- Negative
- 0
- 1
- 2
- > 2

output
- Null
- Empty
- Single character
- Multiple characters

Specification-Based Testing in a Nutshell

```
public static String repeat(final String str, final int repeat)
```

Step 4: Analyze the boundaries

str
- Null string
- Empty string
- String of length 1
- String of length 2
- String of length > 2

repeat
- Negative
- 0
- 1
- 2
- > 2

On and off-points?

2,3

Specification-Based Testing in a Nutshell

```
public static String repeat(final String str, final int repeat)
```

Step 4: Analyze the boundaries

str
- Null string
- Empty string
- String of length 1
- String of length 2
- String of length 3

repeat
- Negative
- 0
- 1
- 2
- 3

On and off-points?

2,3

Specification-Based Testing in a Nutshell

```
public static String repeat(final String str, final int repeat)
```

Step 4: Analyze the boundaries

str
- Null string
- Empty string
- String of length 1
- String of length 2
- String of length 3

repeat
- Negative
- 0
- 1
- 2
- 3

On and off-points?

-1, 0

Specification-Based Testing in a Nutshell

```
public static String repeat(final String str, final int repeat)
```

Step 4: Analyze the boundaries

str
- Null string
- Empty string
- String of length 1
- String of length 2
- String of length 3

repeat
- -1
- 0
- 1
- 2
- 3

On and off-points?

-1, 0

Specification-Based Testing in a Nutshell

```
public static String repeat(final String str, final int repeat)
```

Step 5: Devise test cases

str
- Null string
- Empty string
- String of length 1
- String of length 2
- String of length 3

repeat
- -1
- 0
- 1
- 2
- 3

How many combinations do we have here?

25!

Can we reduce some?

Specification-Based Testing in a Nutshell

```
public static String repeat(final String str, final int repeat)
```

Step 5: Devise test cases

str
- Null string
- Empty string
- String of length 1
- String of length 2
- String of length 3

repeat
- -1
- 0
- 1
- 2
- 3

Special cases

Test case	str	repeat	output
T1	null	2	null
T2	“”	3	“”
T3	“x”	-1	“”
T4	“xx”	0	“”

Specification-Based Testing in a Nutshell

```
public static String repeat(final String str, final int repeat)
```

Step 5: Devise test cases

str
- Null string
- Empty string
- String of length 1
- String of length 2
- String of length 3

repeat
- Negative
- 0
- 1
- 2
- 3

str: String of length 1

Test case	str	repeat	output
T5	“a”	1	“a”
T6	“b”	2	“bb”
T7	“c”	3	“ccc”

Specification-Based Testing in a Nutshell

```
public static String repeat(final String str, final int repeat)
```

Step 5: Devise test cases

str
- Null string
- Empty string
- String of length 1
- String of length 2
- String of length 3

repeat
- -1
- 0
- 1
- 2
- 3

str: String of length 2

Test case	str	repeat	output
T8	“aa”	1	“aa”
T9	“bb”	2	“bbbb”
T10	“cc”	3	“cccc”

Specification-Based Testing in a Nutshell

```
public static String repeat(final String str, final int repeat)
```

Step 5: Devise test cases

str
- Null string
- Empty string
- String of length 1
- String of length 2
- String of length 3

repeat
- -1
- 0
- 1
- 2
- 3

str: String of length 3

Test case	str	repeat	output
T11	“aaa”	1	“aaa”
T12	“bbb”	2	“bbbbbb”
T13	“ccc”	3	“ccccccccc”

Specification-Based Testing in a Nutshell

```
public static String repeat(final String str, final int repeat)
```

Step 6: Automate the test cases

```
@Test
void strIsNull() {
    assertThat(repeat(null, null, 2)).isEqualTo(null);
}
```

JUnit also offers the ParameterizedTest

Specification-Based Testing in a Nutshell

```
class StringUtilitiesTest {

    @ParameterizedTest
    @MethodSource("generator")
    void repeat(String description, String originalString, int times, String expectedString) {
        assertThat(StringUtilities.repeat(originalString, times)).isEqualTo(expectedString);
    }

    static Stream<Arguments> generator() {
        Arguments tc1 = Arguments.of("str null", null, 2, null);
        Arguments tc2 = Arguments.of("empty string", "", 3, "");
        Arguments tc3 = Arguments.of("repeat negative", "x", -1, "");
        Arguments tc4 = Arguments.of("zero repetitions", "xx", 0, "");
        Arguments tc5 = Arguments.of("1 char, 1 repetition", "a", 1, "a");
        Arguments tc6 = Arguments.of("1 char, 2 repetitions", "b", 2, "bb");
        Arguments tc7 = Arguments.of("1 char, N repetitions", "c", 3, "ccc");
        Arguments tc8 = Arguments.of("2 chars, 1 repetition", "aa", 1, "aa");
        Arguments tc9 = Arguments.of("2 chars, 2 repetitions", "bb", 2, "bbbb");
        Arguments tc10 = Arguments.of("2 chars, N repetitions", "cc", 3, "cccccc");
        Arguments tc11 = Arguments.of("3+ chars, 1 repetition", "aaa", 1, "aaa");
        Arguments tc12 = Arguments.of("3+ chars, 2 repetitions", "bbb", 2, "bbbbbb");
        Arguments tc13 = Arguments.of("3+ chars, N repetitions", "ccc", 3, "ccccccccc");
        return Stream.of(tc1, tc2, tc3, tc4, tc5, tc6, tc7, tc8, tc9, tc10, tc11, tc12, tc13);
    }
}
```

Specification-Based Testing in a Nutshell

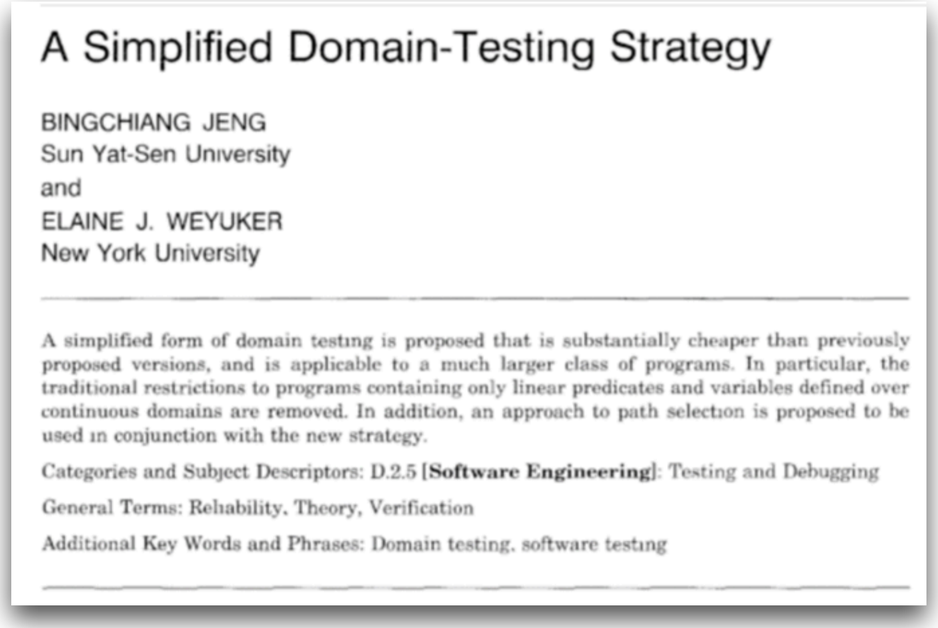
```
public static String repeat(final String str, final int repeat)
```

Step 7: Augment the test suite with creativity and experience

Being systematic is good, but we should never discard our experience.

Multiple Boundaries?

- Handle boundaries independently
- For each boundary, pick on and off point
- While testing one boundary, use varying in points for the remaining boundaries.
- Use domain matrix.



A Simplified Domain-Testing Strategy

BINGCHIANG JENG
Sun Yat-Sen University
and
ELAINE J. WEYUKER
New York University

A simplified form of domain testing is proposed that is substantially cheaper than previously proposed versions, and is applicable to a much larger class of programs. In particular, the traditional restrictions to programs containing only linear predicates and variables defined over continuous domains are removed. In addition, an approach to path selection is proposed to be used in conjunction with the new strategy.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging

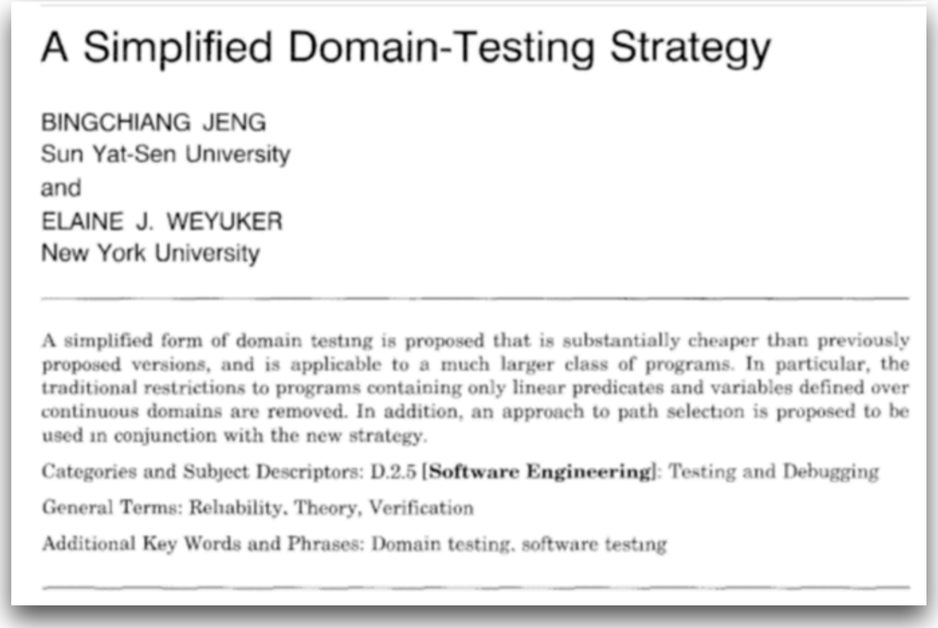
General Terms: Reliability, Theory, Verification

Additional Key Words and Phrases: Domain testing, software testing

Bingchiang Jeng and Elaine J. Weyuker. 1994. "*A simplified domain-testing strategy*". ACM Trans. Softw. Eng. Methodol. 3, 3 (July 1994), 254–270.

Multiple Boundaries?

- Handle boundaries independently
- For each boundary, pick on and off point
- While testing one boundary, use varying in points for the remaining boundaries.
- Use domain matrix.



A Simplified Domain-Testing Strategy

BINGCHIANG JENG
Sun Yat-Sen University
and
ELAINE J. WEYUKER
New York University

A simplified form of domain testing is proposed that is substantially cheaper than previously proposed versions, and is applicable to a much larger class of programs. In particular, the traditional restrictions to programs containing only linear predicates and variables defined over continuous domains are removed. In addition, an approach to path selection is proposed to be used in conjunction with the new strategy.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms: Reliability, Theory, Verification

Additional Key Words and Phrases: Domain testing, software testing

Bingchiang Jeng and Elaine J. Weyuker. 1994. "*A simplified domain-testing strategy*". ACM Trans. Softw. Eng. Methodol. 3, 3 (July 1994), 254–270.

Multiple Boundaries?

```
X > 0 && x <= 10 && y >= 1.0
```

Boundary conditions for " x > 0 && x <= 10 && y >= 1.0"								
			test cases (x, y)					
Variable	**Condition**	**type**	t1	t2	t3	t4	t5	t6
x	> 0	on						
		off						
	<= 10	on						
		off						
	typical	in						
y	>= 1.0	on						
		off						
	typical	in						

Multiple Boundaries?

```
X > 0 && x <= 10 && y >= 1.0
```

Boundary conditions for " x > 0 && x <= 10 && y >= 1.0"								
			test cases (x, y)					
Variable	**Condition**	**type**	t1	t2	t3	t4	t5	t6
x	> 0	on	0					
		off		1				
	<= 10	on			10			
		off				11		
	typical	in						
y	>= 1.0	on					1.0	
		off						0.9
	typical	in						

- Handle boundaries independently
- For each boundary, pick on and off point

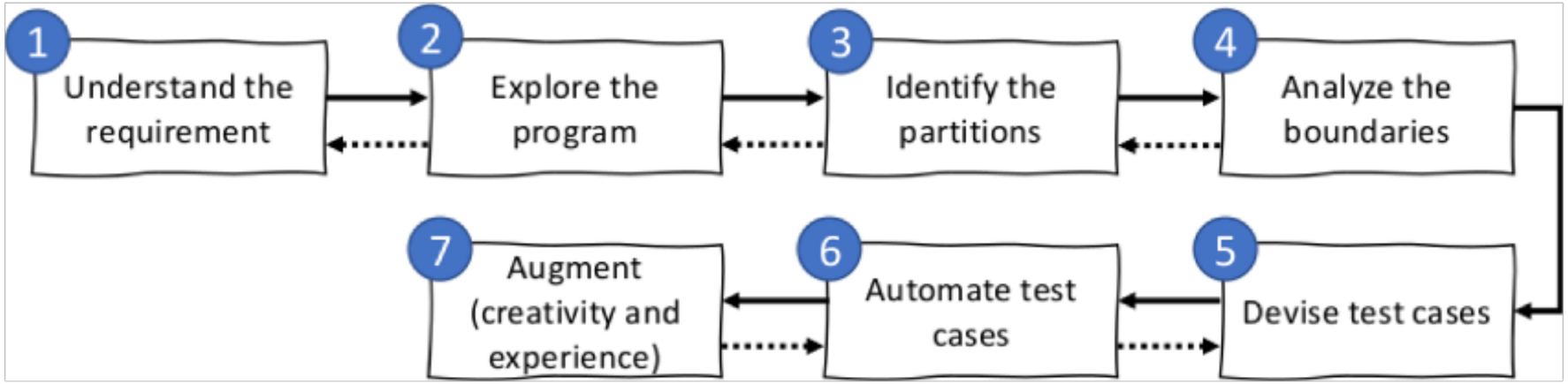
Multiple Boundaries?

```
X > 0 && x <= 10 && y >= 1.0
```

Boundary conditions for " x > 0 && x <= 10 && y >= 1.0"								
			test cases (x, y)					
Variable	**Condition**	**type**	t1	t2	t3	t4	t5	t6
x	> 0	on	0					
		off		1				
	<= 10	on			10			
		off				11		
	typical	in					4	6
y	>= 1.0	on					1.0	
		off						0.9
	typical	in	10.0	16.0	109.3	239.2		

- While testing one boundary, use varying in points for the remaining boundaries.

Specification-Based Testing in a Nutshell



Specification-Based Testing in a Nutshell

Let's practice!

Method: intersection

This method returns a new list containing all elements that are contained in both given lists.

Inputs:
- List1, a list of integers
- List2, a list of integers

Outputs:
- A new list.

- In groups of 2, identify the partitions.
- Time: 5 minutes

Specification-Based Testing in a Nutshell

One possible way to reason about it

- First, each input in isolation:
  - List1 and List2 are both lists. A few straightforward test cases:
    - Null
    - 0 elements
    - 1 element
    - Multiple elements
  - The requirements don't mention any specific constraints (e.g., max size of the list, range of the numbers in the list, etc), so we continue

Specification-Based Testing in a Nutshell

One possible way to reason about it

- First, each input in isolation:
  - List1 and List2 are both lists. A few straightforward test cases:
    - Null
    - 0 elements
    - 1 element
    - Multiple elements
  - The requirements don't mention any specific constraints (e.g., max size of the list, range of the numbers in the list, etc), so we continue
- How inputs interact:
  - The requirement is all about items in both sets, therefore:
    - List1 and List2 have elements in common
      - 1 element
      - More than 1 element
    - List1 and List2 don't have elements in common

Specification-Based Testing in the Real World

- The process should be iterative, not sequential.

Specification-Based Testing in the Real World

- The process should be iterative, not sequential.
- How far should specification testing go? Think about the cost of a failure!

Specification-Based Testing in the Real World

- The process should be iterative, not sequential.
- How far should specification testing go? Think about the cost of a failure!
- Use variations of the same input to facilitate understanding.

Test case	str	repeat	output
T5	“a”	1	“a”
T6	“b”	2	“bb”
T7	“c”	3	“ccc”

Test case	str	repeat	output
T5	“a”	1	“a”
T6	“a”	2	“aa”
T7	“a”	3	“aaa”

Specification-Based Testing in the Real World

- The process should be iterative, not sequential.
- How far should specification testing go? Think about the cost of a failure!
- Use variations of the same input to facilitate understanding.
- When the number of combinations explodes, be pragmatic.

Specification-Based Testing in the Real World

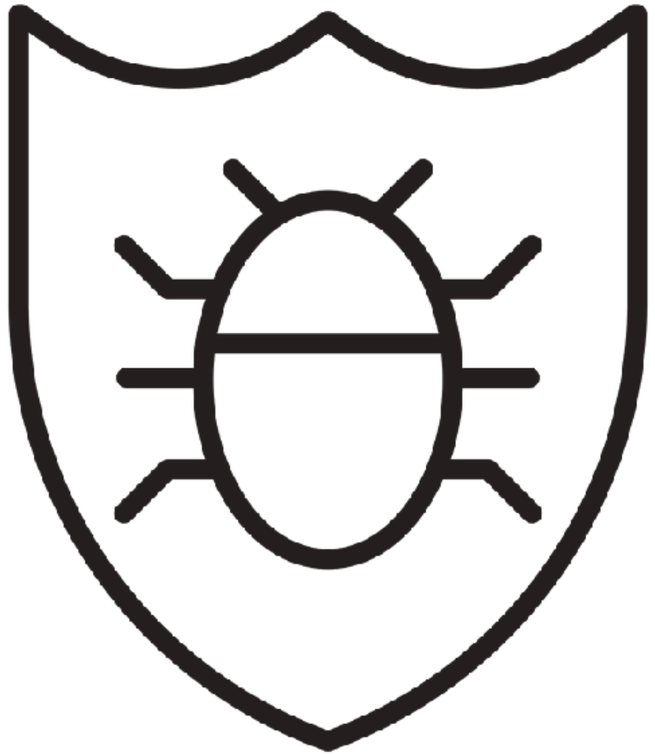
- The process should be iterative, not sequential.
- How far should specification testing go? Think about the cost of a failure!
- Use variations of the same input to facilitate understanding.
- When the number of combinations explodes, be pragmatic.
- When in doubt, go for the simplest input.

Specification-Based Testing in the Real World

- The process should be iterative, not sequential.
- How far should specification testing go? Think about the cost of a failure!
- Use variations of the same input to facilitate understanding.
- When the number of combinations explodes, be pragmatic.
- When in doubt, go for the simplest input.
- Test for nulls and exceptional cases, but only when it makes sense.

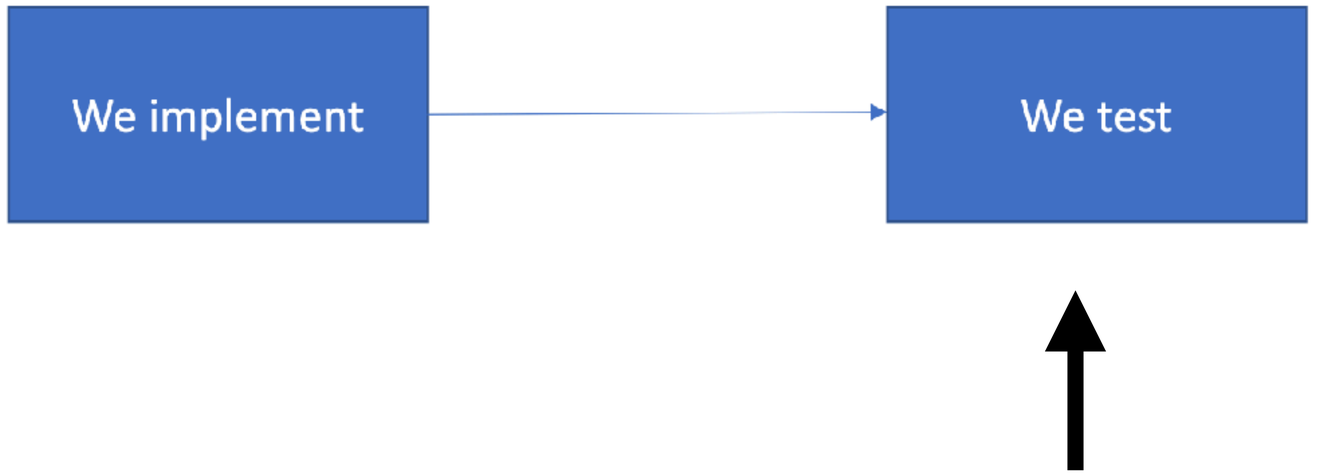
Software Verification

Test-Driven Development



Bin Lin

What We Do Currently…



What We Do Currently…

Can we test first?

- We think about a test.
- We write the test.
- We implement the code …
- In the simplest way we can.

Let’s Try!

Convert Roman numerals to Hindu-Arabic numeral system

- “I” -> 1
- “III” -> 3
- “VI” -> 6
- “IV” -> 4
- “XVI” -> 16
- “XIV” -> 14
- …

Baby Steps

Simplicity: We should do the simplest implementation that solves the problem, start by the simplest possible test…

- “I” -> 1

```
public class RomanNumberConverterTest {
  @Test
  void shouldUnderstandSymbolI() {
    RomanNumeralConverter roman = new RomanNumeralConverter();
    int number = roman.convert("I");
    assertThat(number).isEqualTo(1);
  }
}
```

We will get compilation errors here, as the RomanNumeralConverter class does not exist!

Baby Steps

Simplicity: We should do the simplest implementation that solves the problem, start by the simplest possible test…

- “I” -> 1

```
public class RomanNumeralConverter {
  public int convert(String numberInRoman) {
    return 1;
  }
}
```

Returning 1 makes the test pass. But is this the implementation we want?

Baby Steps

Simplicity: We should do the simplest implementation that solves the problem, start by the simplest possible test…

- “I” -> 1
- **“III” -> 3**

```
public class RomanNumeralConverter {
  public int convert(String numberInRoman) {
    return 1;
  }
}
```

Returning 1 makes the test pass. But is this the implementation we want?

Baby Steps

Simplicity: We should do the simplest implementation that solves the problem, start by the simplest possible test…

- “I” -> 1
- “III” -> 3

Kent Beck states in his book: “*Do these steps seem too small to you? Remember, TDD is not about taking teensy tiny steps, it’s about being able to take teensy tiny steps. Would I code day-to-day with steps this small? No. But when things get the least bit weird, I’m glad I can.*”

Refactor!

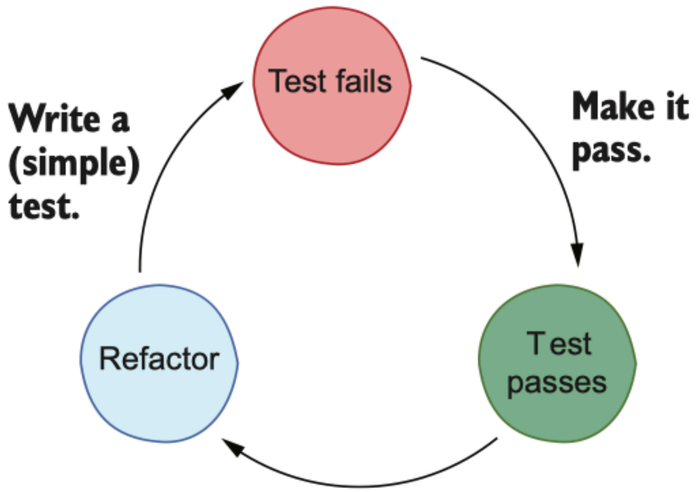
In many opportunities, we are so busy making the test pass that we forget about writing good code.

- After the test passes, you can refactor.
- Good thing is that, after the refactoring, tests should still pass.

Refactoring can be at low-level or high-level.

- Low-level: rename variables, extract methods
- High-level: change the class design

Test-Driven Development



red-green-refactor cycle

We wrote a (unit) test for the next piece of functionality we wanted to implement. The test failed.

We implemented the functionality. The test passed.

We refactored our production and test code.

Advantages of Test-Driven Development

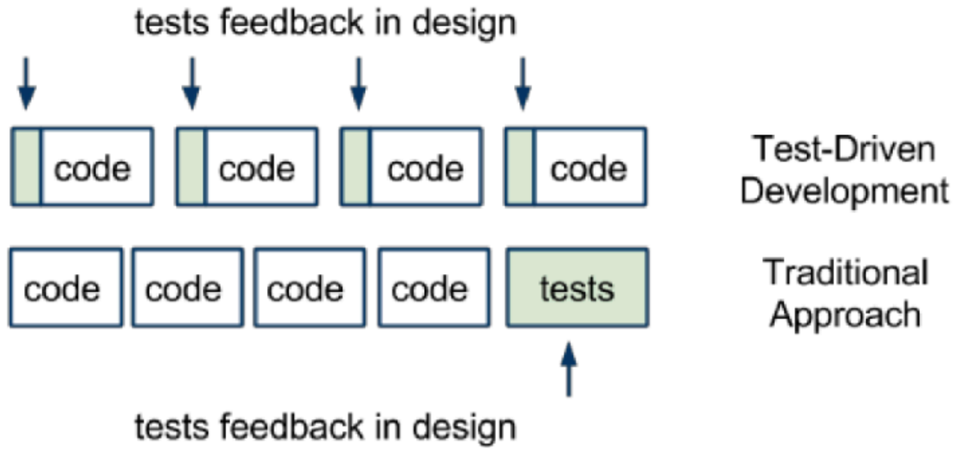
- Looking at the requirements first

Advantages of Test-Driven Development

- Looking at the requirements first
- Full control over the pace of writing production code

Advantages of Test-Driven Development

- Looking at the requirements first
- Full control over the pace of writing production code
- Quick feedback



Advantages of Test-Driven Development

- Looking at the requirements first
- Full control over the pace of writing production code
- Quick feedback
- Testable code

Advantages of Test-Driven Development

- Looking at the requirements first
- Full control over the pace of writing production code
- Quick feedback
- Testable code
- Feedback about design

Is It Really Effective?

- 50% more tests, less time debugging [Janzen].
- 40-50% less defects, no impact on productivity [Maximilien & Williams].
- 40-50% less defects in Microsoft and IBM products [Nagappan & Bhat].
- Better use of OOP concepts [Janzen & Saiedian].
- More cohesive, less coupled [Steinberg].

Janzen, D., "*Software Architecture Improvement through Test-Driven Development*". Conference on Object Oriented Programming Systems Languages and Applications, ACM, 2005.

Maximilien, E. M. and L. Williams. "*Assessing test-driven development at IBM*". IEEE 25th International Conference on Software Engineering, Portland, Orlando, USA, IEEE Computer Society, 2003.

Nagappan, N., Bhat, T. "*Evaluating the efficacy of test- driven development: industrial case studies*". Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering.

Janzen, D., Saiedian, H. "*On the Influence of Test-Driven Development on Software Design*". Proceedings of the 19th Conference on Software Engineering Education & Training (CSEET'06).

Steinberg, D. H. "*The Effect of Unit Tests on Entry Points, Coupling and Cohesion in an Introductory Java Programming Course*". XP Universe, Raleigh, North Carolina, USA, 2001.

Is It?

- No difference in code quality [Erdogmus et al., Müller et al.]
- The differences in the program code, between TDD and the iterative test-last development, were not as clear as expected [Siniaalto & Abrahamsson].

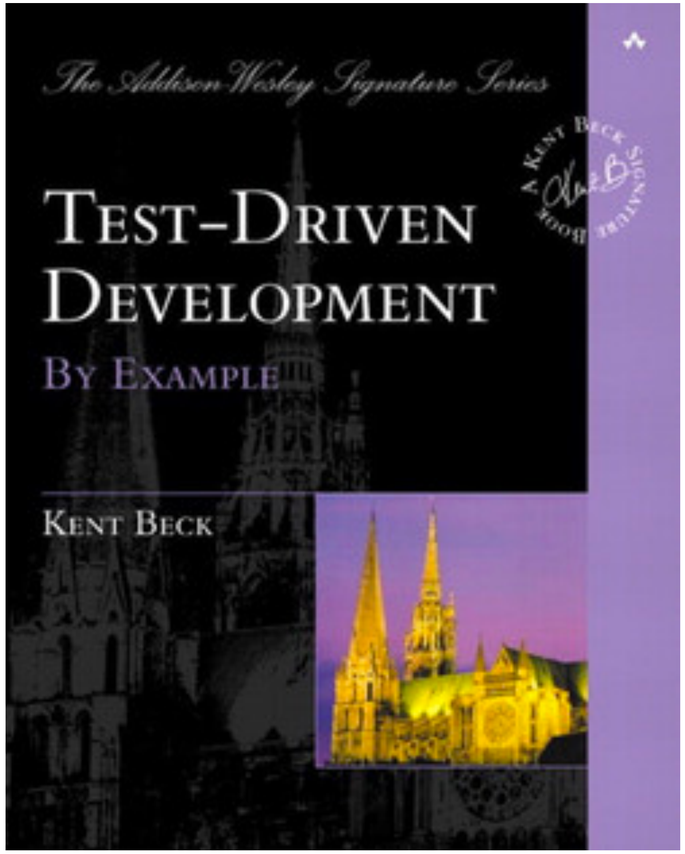
Erdogmus, H., Morisio, M., et al. "*On the effectiveness of the test-first approach to programming*". IEEE Transactions on Software Engineering 31(3): 226 – 237, 2005.
Müller, M. M., Hagner, O. "*Experiment about test-first programming*". IEE Proceedings 149(5): 131 – 136, 2002.
Siniaalto, Maria, and Pekka Abrahamsson. "*Does test-driven development improve the program code? Alarming results from a comparative case study*." Balancing Agility and Formalism in Software Engineering. Springer Berlin Heidelberg, 2008. 143-156.

Is It?

- *"The practice of test-driven development does not drive directly the design, but gives them a safe space to think, the opportunity to refactor constantly, and subtle feedback given by unit tests, are responsible to improve the class design".*
- *"The claimed benefits of TDD may not be due to its distinctive test-first dynamic, but rather due to the fact that TDD-like processes encourage finegrained, steady steps that improve focus and flow."*

Aniche, M., & Gerosa, M. A. (2015). "*Does test-driven development improve class design? A qualitative study on developers' perceptions*". Journal of the Brazilian Computer Society, 21(1), 15.
Fucci, D., Erdogmus, H., Turhan, B., Oivo, M., & Juristo, N. (2016). "*A Dissection of Test-Driven Development: Does It Really Matter to TestFirst or to Test-Last?*". IEEE Transactions on Software Engineering.

More About TDD



Test Driven Development: By Example

by Kent Beck
Released November 2002
Publisher(s): Addison-Wesley Professional
ISBN: 9780321146533

Credits

- Chapter 1, 2, 8, Effective Software Testing by Maurício Aniche, https://www.effective-software-testing.com/
- Chapter 1, Introduction to Software Testing (Edition 2) by Paul Ammann and Jeff Offutt
- Chapter 1, Software Testing: A Craftsman's Approach (Fifth Edition) by Paul C. Jorgensen and Byron DeVries
- CSE1110 - Software Quality and Testing by Arie van Deursen and Maurício Aniche, https://se.ewi.tudelft.nl/cse1110-2019/