Software Testing & Unit Test Frameworks

Jieung Kim

Aug. 2022

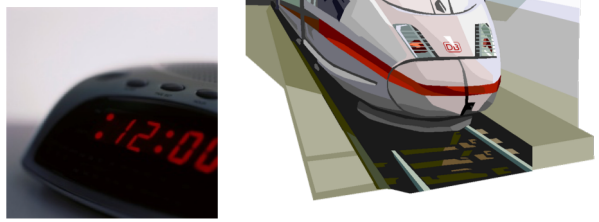
Contents

Contents

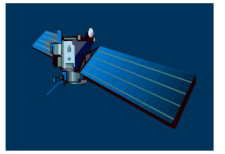
- Software testing
  - Why testing is necessary
  - Kinds of testing
- Introduce Googletest framework
  - Googletest framework – Unit testing framework in C++
  - Example
- Introduce Python unittest framework
  - Python unittest framework – Unit testing library in Python
  - Example

Software testing

Software in the world

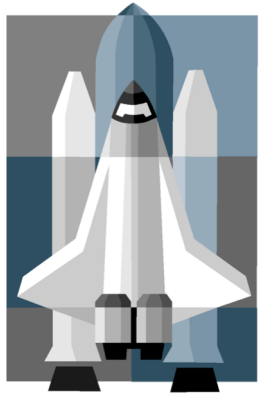
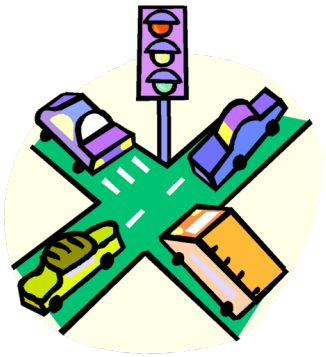




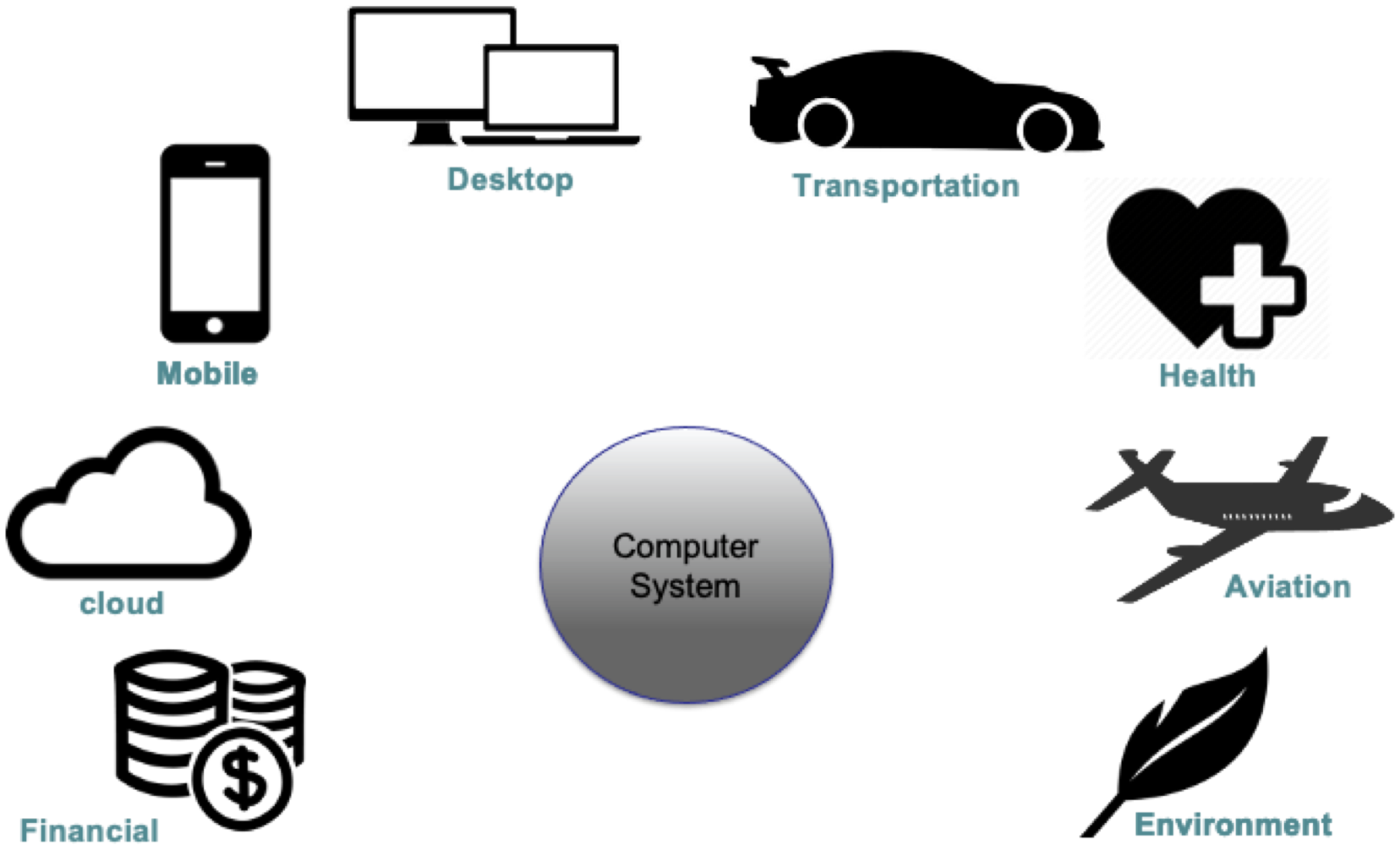






Software crisis



Software crisis

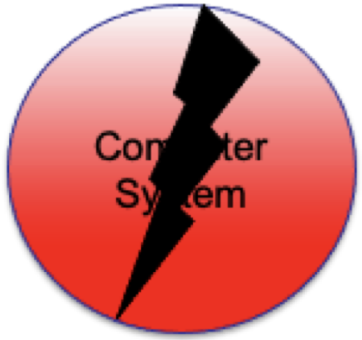
Crash

Accident

Mobile

Life

cloud



Loss

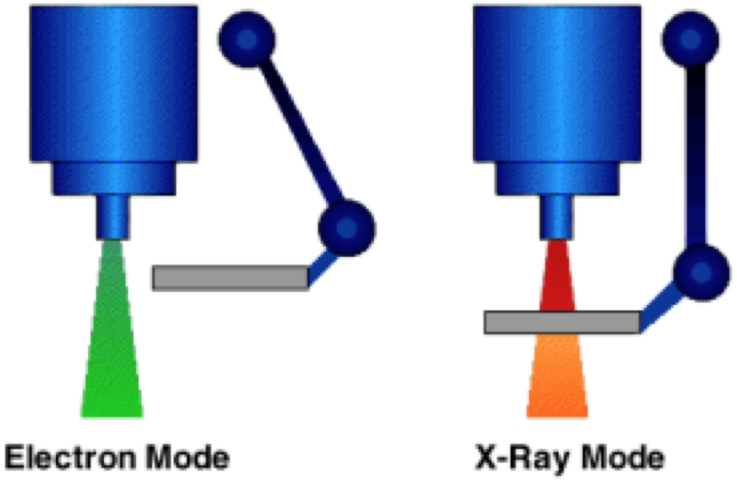
Financial

Environment

Software crisis

Radiotherapeutic medical device

- Derived from Therac-6
  - Two basic modes of operation
  - Safety features in hardware instead of software
- 6 confirmed deaths with a root cause of radiation burns
  - Software race condition
  - Poor software design and QA
  - Misleading user interface
  - Root cause: Poor understanding of software reliability issue



Software crisis

Ariane 5

- Derived from Ariane 4 (reuses code from previous reliable and time-prove vehicle)
- Exploded on its first voyage on June 4th 1996
  - 64 bit float containing velocity truncated to a 16 bit integer in a non-critical software component
    - Caused an uncaught exception that propagated to the control component
    - A safety component triggered mission abort
    - The non-critical component served no actual purpose
- $370 million in damage
- ESA had spent 10 years and $7 billion developing the A5

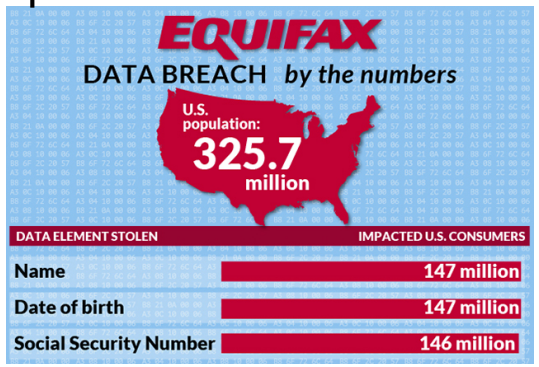
Software failure

Ariane 5 explosion
$370 million

1996

...

50% of American personal record



2018

Recalls More than 150,000 vehicles

2021~2022

Software failure

AUTHOR

HERB KRASNER

CISQ Advisory Board Member and retired Professor of Software Engineering at the University of Texas at Austin.

He can be reached at hkrasner@utexas.edu.

THE COST OF POOR SOFTWARE QUALITY IN THE US: A 2020 REPORT

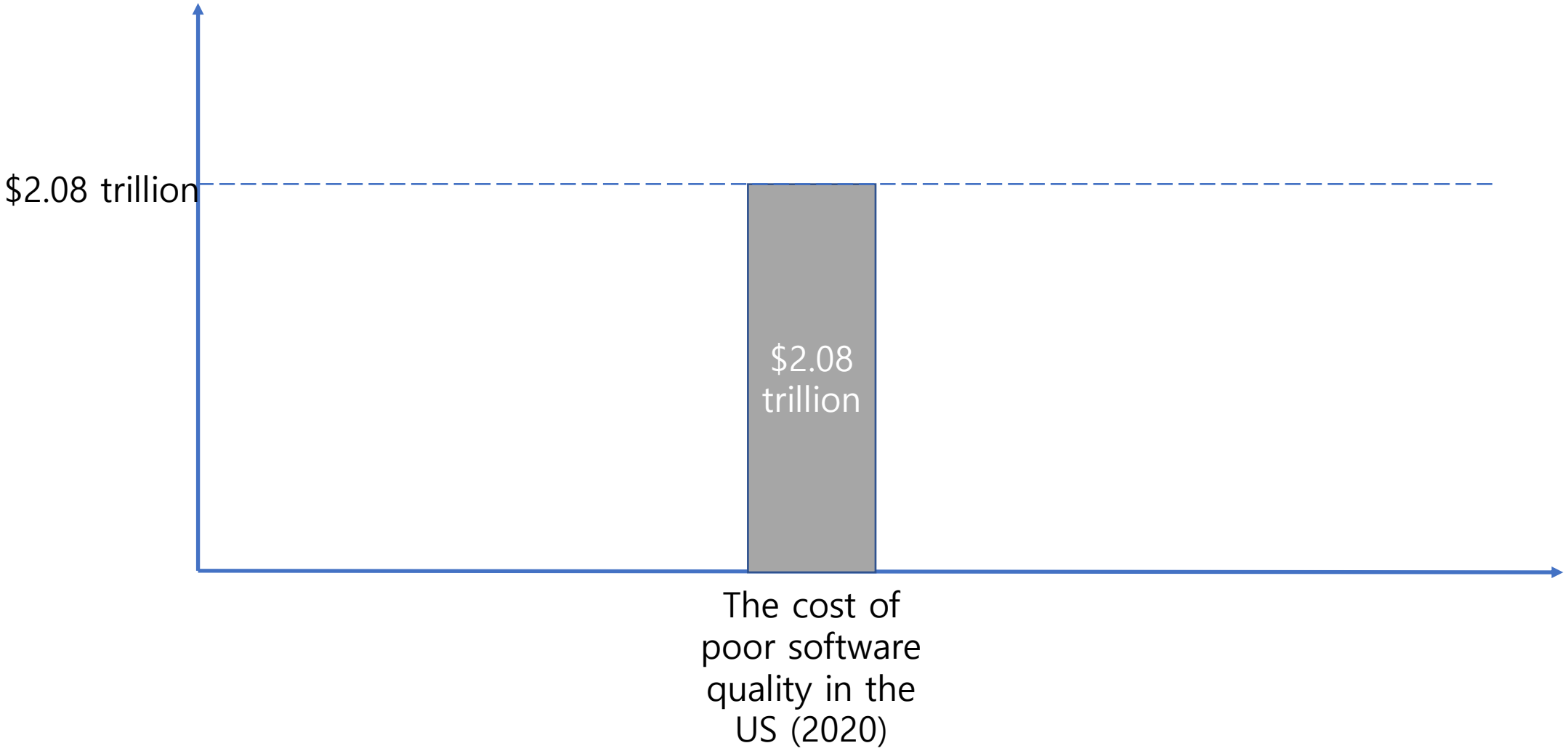
SYNOPSYS®
Silicon to Software™

Software Engineering Institute
Carnegie Mellon

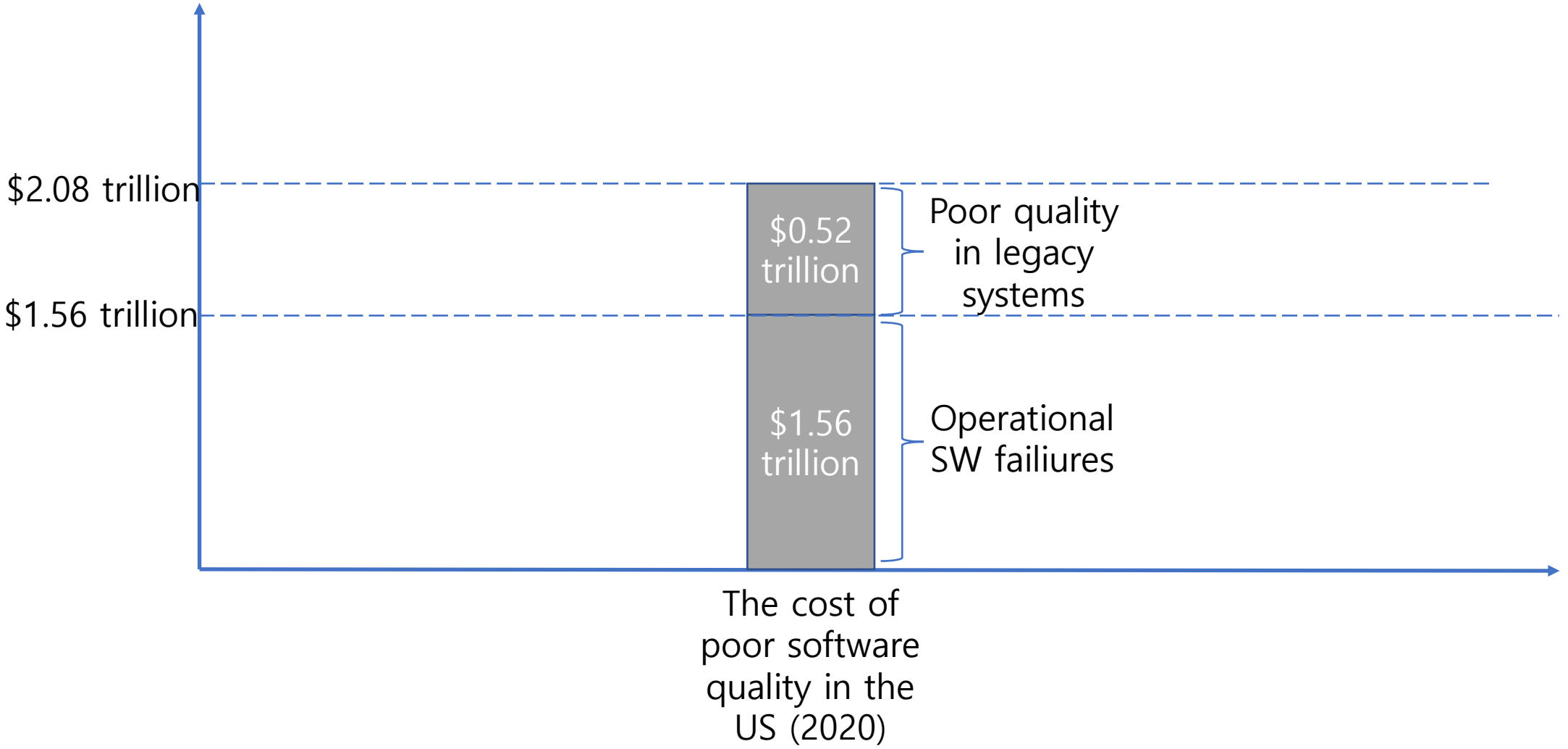
NORTHROP GRUMMAN

- Unsuccessful IT/software projects - $260 billion (up from $177.5 billion in 2018)
- Poor quality in legacy systems - $520 billion (down from $635 billion in 2018)
- Operational software failures - $1.56 trillion (up from $1.275 trillion in 2018)

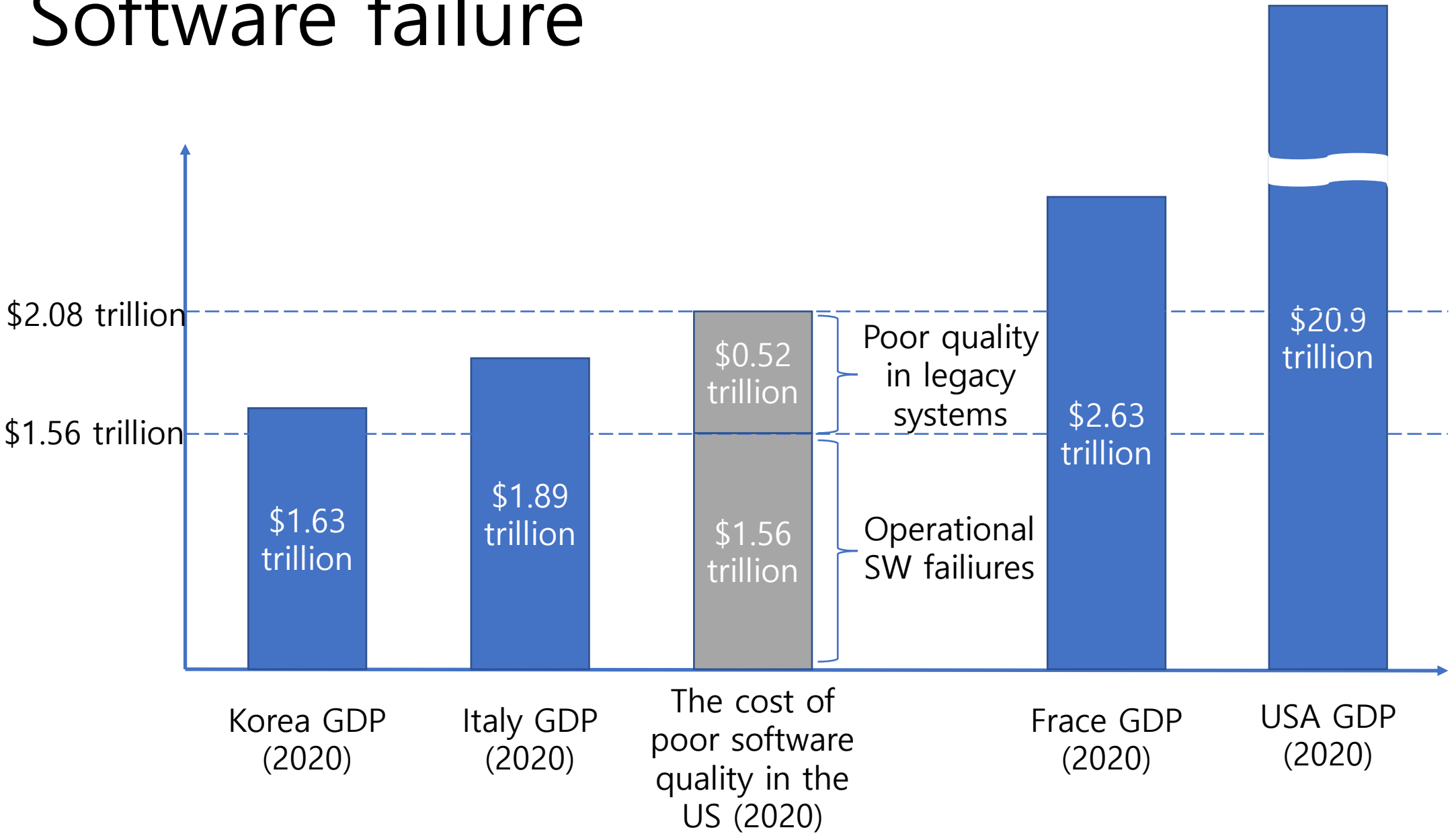
Software failure



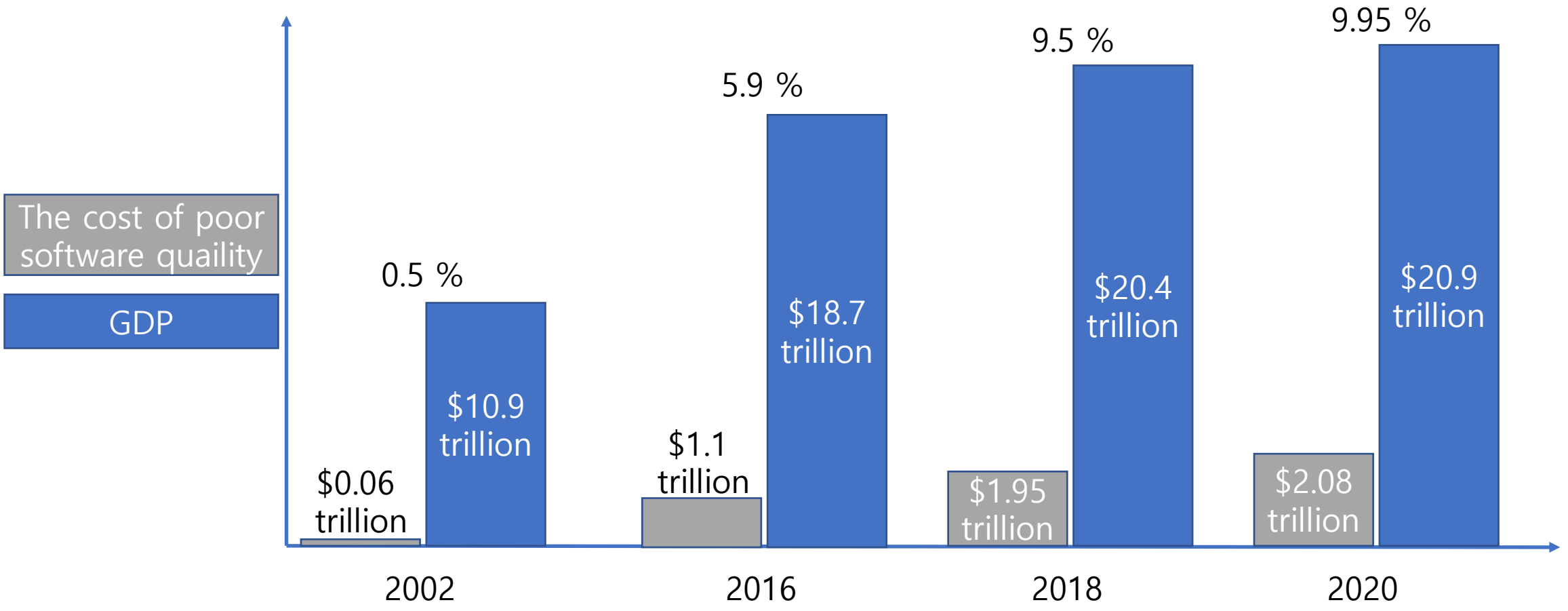
Software failure



Software failure



Software failure



Why do these things happen? - bugs

A software bug

- Is an **error, flaw, or fault** in a computer program or system.
- **Causes** the software to produce an **incorrect or unexpected result**, or to **behave in unintended ways**
- Some are minor, but others **cause disasters**
- As we discussed, the **cost is huge**

Source of bugs

- **Arithmetic**
  - Division by zero
  - Arithmetic overflow or underflow
  - Loss of arithmetic precision
    - Rounding
    - Numerically unstable algorithms (e.g., using floating point operations)
- **Logic**
  - Infinite loops and infinite recursion
- **Recourse**
  - Null pointer dereference
  - Buffer overflow
  - Double free error
  - Access violation

Source of bugs

- **Interface**
  - Incorrect API usage
  - Incorrect hardware handling
  - Incorrect assumptions of a particular platform
- **Teamworking**
  - Unpropagated updates
  - Comments out of data or incorrect
  - Differences between documentation and product

How can we prevent bugs?

- Software engineering approach
  - E.g., Software development life cycle / code review
- Programming language support
  - E.g., type checker
- Testing
- Static analysis
- Verification
- Etc…

Software testing

Evaluation of the software against requirements gathered fro m users and system specifications.

Fault: Should start searching at `0`, not `1`

```
// Effects: If arr is null throw exception
// else return the number of occurrences of 0 in arr
int numZero (std::vector<int> arr) {
    int count = 0;
    for (int i = 1; i < arr.size(); i++) {
        if (arr [ i ] == 0) {
            count++;
        }
    }
    return count;
}
```

Error: `i` is `1`, not `0`, on the first iteration
Failure: none

Test 1
`[2,7,0]`
Expected: `1`
Actual: `1`

Test 2
`[0,2,7]`
Expected: `1`
Actual: `0`

Error: `i` is `1`, not `0`
Error propagates to the variable count
Failure: count is `0` at the return statement

Does testing work?

- "measuring over 20 projects: if you have a large number of u nit tests your code will be **an order of magnitude (x10)** less complex."
- **Controlled study results**:
  - "..quality increased linearly with the number of programmer tests..."
  - "..test-first students on average wrote more tests and, in turn, students who wrote more tests tended to be more productive..."

http://agilepainrelief.com/notesfromatooluser/2008/11/misconceptions-with-test-driven-development.html
http://collaboration.csc.ncsu.edu/laurie/Papers/TDDpaperv8.pdf

Testing in the industry

- Is testing actually and actively used in the industry?

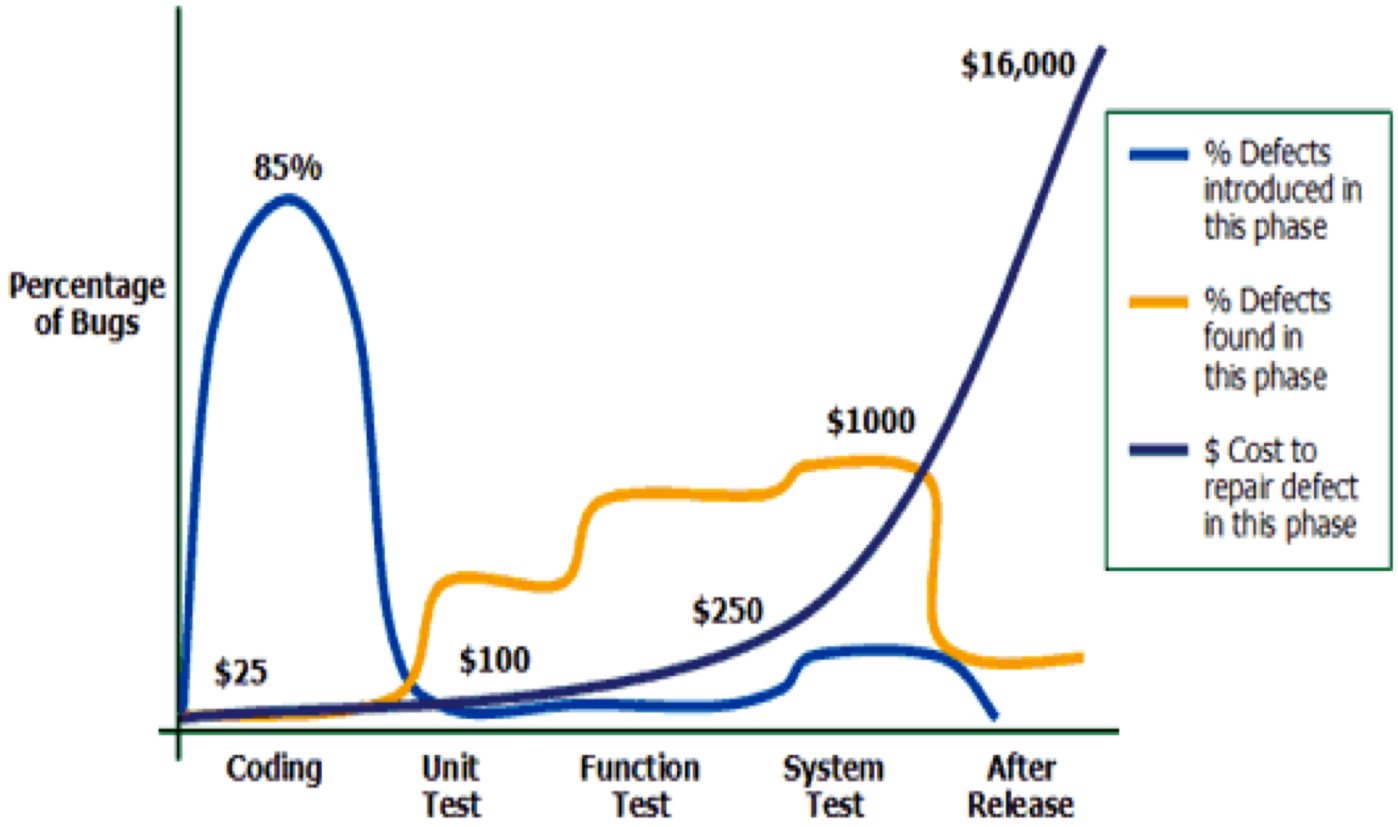
Definitely YES

- If there is a company that does not actively test their product with multiple levels, **DO NOT USE** their products and **DO NO T WORK** for them

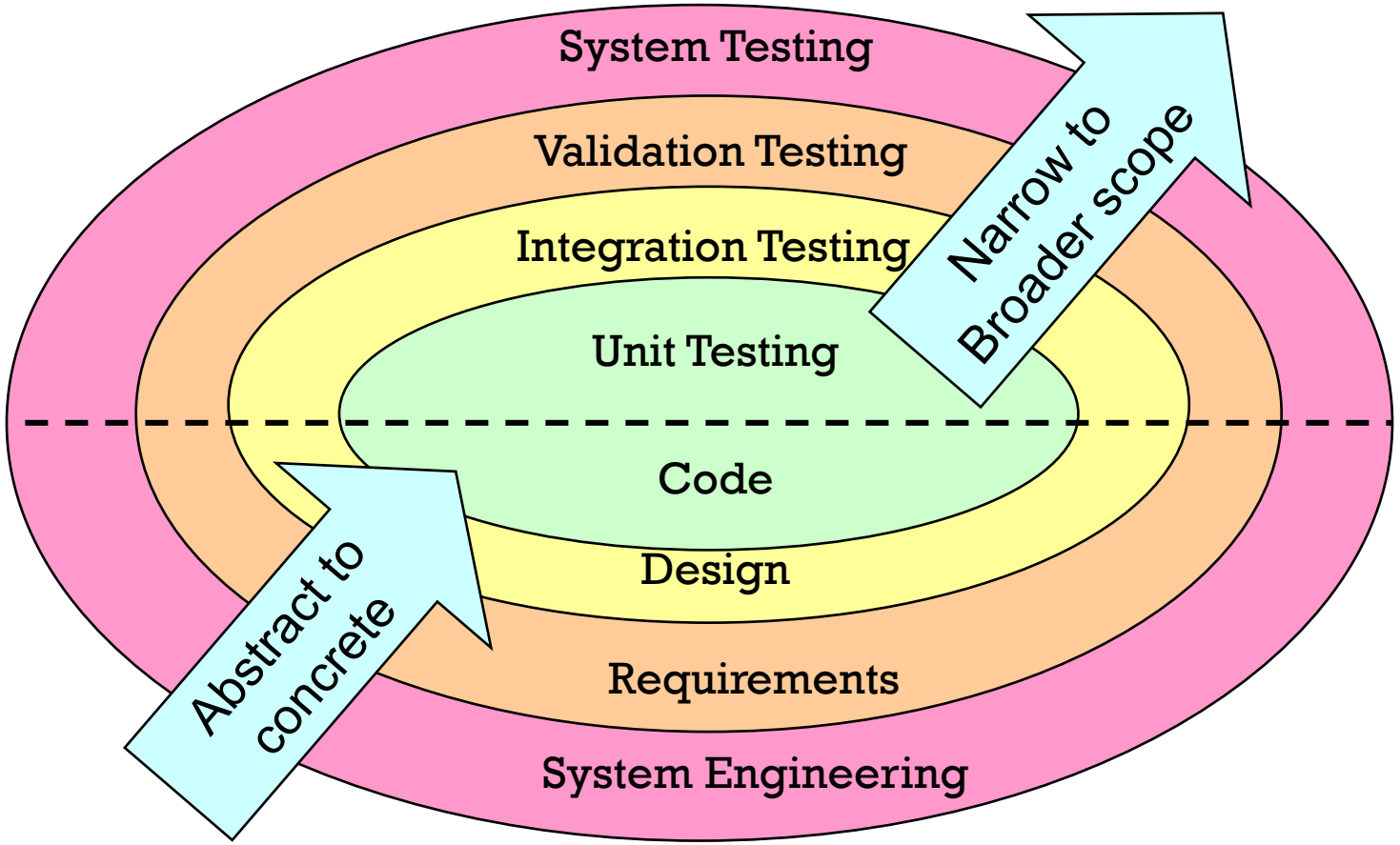
Is testing used in the industry?

- Software testing engineer
  - Responsible for **designing and implementing test procedures** to ensure that software programs work as intended
  - Mostly hired by **software development companies**
    - Ensure that **products perform to specifications** before being released
- Software test from software engineer
  - Google recommends a certain coverage with unit tests for the team's code
    - When we code something, we usually make suitable tests together
  - Google development tools also provide several testing when before/after we submit our codes

Testing – when do we need to test?



Testing – levels



Testing – levels

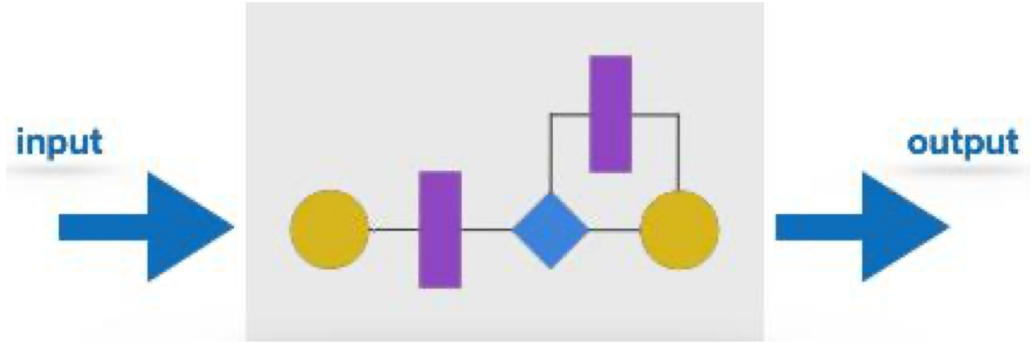
- **Unit testing**
  - Concentrates on each component/function of the software as implemented in the source code
- **Integration testing**
  - Focuses on the design and construction of the software architecture
- **Validation testing**
  - Requirements are validated against the constructed software
- **System testing**
  - The software and other system elements are tested as a whole

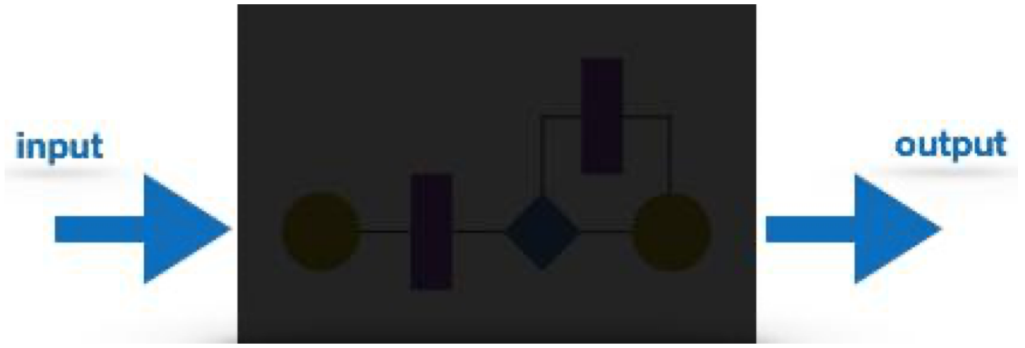
Testing – levels

- **Unit testing**
  - Exercises specific paths in a component's control structure to ensure complete coverage and maximum error detection
  - Components are then assembled and integrated
- **Integration testing**
  - Focuses on inputs and outputs, and how well the components fit together and work together
- **Validation testing**
  - Provides final assurance that the software meets all functional, behavioral, and performance requirements
- **System testing**
  - Verifies that all system elements (software, hardware, people, databases) mesh properly and that overall system function and performance is achieved

Testing – approaches

- **The "box" approach**
  - **White-box testing**
    - Uses the control structure part of component-level design to derive the test cases
  - **Black-box testing**
    - Focuses on the functional requirements and the information domain of the software
    - The tester identifies a set of input conditions that will fully exercise all functional requirements for a program
- Static, dynamic, etc.





Testing – types, techniques, and tactics

- Alpha testing
  - Carried out by the test team within the developing organization
- Beta testing
  - Performed by a selected group of friendly customers
- Acceptance testing
  - Performed by the customer to determine whether to accept or reject the delivery of the system
- Performance testing, stress testing, volume testing, config. testing, compatibility testing, regression testing, maintenance testing, usability testing, etc

Testing – unit testing tools

- Java
  - NUnit, Junit, TestNG, Mockito, and PHPUnit
- Python
  - Robot, PyTest, Unittest, DocTest, Nose2, and Testify
- C/C++
  - Googletest, Boot Test Library, QA Systems Cantata, Parasoft C/C++ test, Microsoft Visual Studio, Cppunit, Catch, Bandit, and CppUTest
- JavaScript
  - Jest, Mocah, Storybook, Jasmine, Cypress, Puppeteer, Testing Library, and WebdriverIO

Googletest framework

Googletest framework overview

- A **unit testing library** for the C++ programming language.
- **Repository**
  - http://code.google.com/p/googletest/
- **Projects using Google Test**
  - Android Open Source Project operating system
  - Chromium projects (behind the Chrome browser, Edge browser, and Chrome OS)
  - LLVM compiler
  - Protocol Buffers (Google's data interchange format)
  - OpenCV computer vision library
  - Several internal C++ projects at Google

Googletest framework overview

- **Study materials**
  - README file: https://github.com/google/googletest/blob/master/README.md
  - Googletest user's guide: https://google.github.io/googletest/
  - Whittaker, James (2012). How Google Tests Software. Boston, Massachusetts: Pearson Education. ISBN 0-321-80302-7.
  - A quick introduction to the Google C++ Testing Framework, Arpan Sen, IBM DeveloperWorks, 2010-05-11

Let's briefly look at this

Then, let's look at the example in Visual Studio 2019

Creating a basic test

- Target code: prototype for square-root

```
double square-root (const double);
```

- Test case with Googletest

```
#include "gtest/gtest.h"

TEST (SquareRootTest, PositiveNos) {
    EXPECT_EQ (18.0, square-root (324.0));
    EXPECT_EQ (25.4, square-root (645.16));
    EXPECT_EQ (50.3321, square-root (2533.310224));
}
TEST (SquareRootTest, ZeroAndNegativeNos) {
    ASSERT_EQ (0.0, square-root (0.0));
    ASSERT_EQ (-1, square-root (-22.0));
}
```

Creating a basic test

```
double square-root (const double);
```

Predefined macro in `gtest.h`

Test hierarchy name

Unit test name

```
#include "gtest/gtest.h"

TEST (SquareRootTest, PositiveNos) {
    EXPECT_EQ (18.0, square-root (324.0));
    EXPECT_EQ (25.4, square-root (645.16));
    EXPECT_EQ (50.3321, square-root (2533.310224));
}
TEST (SquareRootTest, ZeroAndNegativeNos) {
    ASSERT_EQ (0.0, square-root (0.0));
    ASSERT_EQ (-1, square-root (-22.0));
}
```

Predefined macros that checks result of `squar e-root` **function**

Assertions

- Google Test assertions are macros that resemble function calls.
- You test a class or function by making assertions about its beha vior.
- `EXPECT_*`
  - Non fatal assertion.
  - Versions generate nonfatal failures.
  - Test will be continued even if the assertion is not satisfied.
- `ASSERT_*`
  - Fatal assertion.
  - Versions generate fatal failures when they fail, and abort the current func tion.
  - Test will directly fail if the assertion is not satisfied.

Assertions

- Basic assertions

Fatal assertion	Nonfatal assertion	Verifies
`ASSERT_TRUE(condtion);`	`EXPECT_TRUE(condtion);`	`Condition` **is true**
`ASSERT_FALSE(condition);`	`EXPECT_FALSE(condition);`	`Condition` **is false**

Assertions

- Binary comparison

Fatal assertion	Nonfatal assertion	Verifies
`ASSERT_EQ(expected, actual);`	`EXPECTED_EQ(expected, actual);`	`expected == actual`
`ASSERT_NE(val1, val2);`	`EXPECT_NE(val1, val2);`	`val1 != val2`
`ASSERT_LT(val1, val2);`	`EXPECT_LT(val1, val2);`	`val1 < val2`
`ASSERT_LE(val1, val2);`	`EXPECT_LE(val1, val2);`	`val1 <= val2`
`ASSERT_GT(val1, val2);`	`EXPECT_GT(val1, val2);`	`val1 > val2`
`ASSERT_GE(val1, val2);`	`EXPECT_GE(val1, val2);`	`val1 >= val2`

Assertions

- Assertions for other types
  - Strings
    - Case sensitive
      - `{ASSERT | EXPECT}_STREQ(str1, str2);`
      - `{ASSERT | EXPECT}_STRNE(str1, str2);`
    - Ignoring case
      - `{ASSERT | EXPECT}_STRCASEEQ(str1, str2);`
      - `{ASSERT | EXPECT}_ STRCASENE(str1, str2);`
  - Double and floating point values
    - `{ASSERT | EXPECT}_FLOAT_EQ(expected, actual);`
    - `{ASSERT | EXPECT}_DOUBLE_EQ(expected, actual);`
    - `{ASSERT | EXPECT}_NEAR (expected, actual, absolute_range);`

Running the test – main function

- Initialize the framework
- Must be called before `RUN_ALL_TESTS`

```
int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

- Must be called only once
  - Multiple calls to it conflicts some features of the framework
- Automatically detects and runs all test tests defined using the `TEST` macro

Running the test – result

```
Running main() from user_main.cpp
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from SquareRootTest
[ RUN      ] SquareRootTest.PositiveNos
..\user_sqrt.cpp(6862): error: Value of: sqrt (2533.310224)
  Actual: 50.332
Expected: 50.3321
[  FAILED  ] SquareRootTest.PositiveNos (9 ms)
[ RUN      ] SquareRootTest.ZeroAndNegativeNos
[       OK ] SquareRootTest.ZeroAndNegativeNos (0 ms)
[----------] 2 tests from SquareRootTest (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (10 ms total)
[  PASSED  ] 1 test.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] SquareRootTest.PositiveNos

1 FAILED TEST
```

Test fixtures

- Help you set up common and custom se ups that tests ne ed.
  - It is typical to do some custom initialization work before executing a unit test.
  - E.g., If you are trying to measure the time/memory footprint of a t est, you need to put some test-specific code in place to measure t hose values.

A test fixture class

```
class myTestFixture1: public ::testing::Test {
public:
    myTestFixture1( ) {
        // initialization code here
    }

    void SetUp( ) {
        // code here will execute just before the test ensues
    }

    void TearDown( ) {
        // code here will be called just after the test completes
        // ok to through exceptions from here if need be
    }

    ~myTestFixture1( ) {
        // cleanup any pending stuff, but no exceptions allowed
    }

    // put in any custom data members that you need
};
```

Defined in `gtest.h`

A test fixture class

- Initialization or allocation
  - In either the constructor or the `SetUp` method.
- Deallocation of resources
  - Either in `TearDown` or the destructor routine.
  - If you want exception handling you must do it only in the `TearDown` code
    - Throwing an exception from the destructor results in undefined behavior.
- Fixture class scope
  - The same test fixture is not used across multiple tests.
  - For every new unit test, the framework creates a new test fixture.

Test with fixture

- Using **`TEST_F`** instead of **`TEST`**.

```
TEST_F (myTestFixture1, UnitTest1) {
    …
}

TEST_F (myTestFixture1, UnitTest2) {
    …
}
```

Advanced features

- More advanced features are available
  - More assertions.
  - Skipping test exectuion.
  - Teaching googletest how to print your values.
  - Death tests
  - Logging additional information
  - Value-parameterized tests

Please look at "Advanced Topics" at "Googletest user's guide".

Python unittest library

Python unittest overview

- A module in the Python standard library that provides various auto mations for testing
- Main concepts
  - **TestCase**: basic unit for tests in the unittest framework
  - **Test suite**: a set of test cases
  - **Fixture**
    - Codes that will be performed before and after test functions
    - It is useful to check whether the testing environment is well established before performing the actual test
    - It is also used to build database or tables and clean up resources before/after testing
  - **Assertion**
    - It determines whether each unit test passes
    - It provides various checkers, including bool tests, validities of instances and exception handlings
    - Tests will fail when assertion fails

Assertions

Statement	Meaning	Statement	Meaning
assertEqual(a, b)	a == b	assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True	assertFalse(x)	bool(x) is False
assertis(a, b)	a is b	assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None	assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b	assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)	assertNotIsInstance(a, b)	not instance(a, b)

Python unittest overview

1. Import unittest module
2. Create a subclass of “unittest.TestCase”
3. Make a test method with the name “test*”. Add self.assert*() to check the result.
4. Call unittest.main() to run the test

Python unittest (simple example)

```
# myCalc.py
def add(a, b):
    return a + b

def substract(a, b
):
    return a - b
```

Python unittest (simple example)

```
# tests.py
import unittest
import myCalc

class MyCalcTest(unittest.TestCase):

    def test_add(self):
        c = myCalc.add(20, 10)
        self.assertEqual(c, 30)

    def test_substract(self):
        c = myCalc.substract(20, 10)
        self.assertEqual(c, 10)

if __name__ == '__main__':
    unittest.main()
```

Python unittest (Test fixture)

```
# myUtil.py
import os

def filelen(filename):
    f = open(filename, "r")
    f.seek(0, os.SEEK_END)
    return f.tell()

def count_in_file(filename, char_to_find):
    count = 0
    f = open(filename, "r")
    for word in f:
        for char in word:
            if char == char_to_find:
                count += 1
    return count
```

Python unittest (Test fixture)

```
import unittest
import os
import myUtil

class MyUtilTest(unittest.TestCase):
  testfile = 'test.txt'

  # Fixture
  def setUp(self):
    f = open(MyUtilTest.testfile, 'w')
      f.write('1234567890')
      f.close()

  def tearDown(self):
    try:
      os.remove(MyUtilTest.testfile)
    except:
      pass
```

```
  def test_filelen(self):
    leng = myUtil.filelen(
           MyUtilTest.testfile)
    self.assertEqual(leng, 10)

  def test_count_in_file(self):
    cnt = myUtil.count_in_file(
          MyUtilTest.testfile, '0')
    self.assertEqual(cnt, 1)

if __name__ == '__main__':
    unittest.main()
```

Conclusion

Conclusion

We looked at the following items

- Software testing
  - Why testing is necessary
  - Kinds of testing
- Introduce Googletest sframework
  - Googletest framework – Unit testing framework for C++
  - Example
- Introduce Python unittest framework
  - Python unittest framework – Unit testing library in Python
  - Example