How a Software Engineer Works?

JIEUNG KIM

jieungkim@inha.ac.kr

인하대학교
INHA UNIVERSITY

Contents

- How a software engineer works
  - Software vs. program
  - Software development life cycle
  - Day in the life of a software engineer
- Two important skills for software engineers
  - Code review
  - Software testing & unit testing

How a software engineer works

Software vs. program

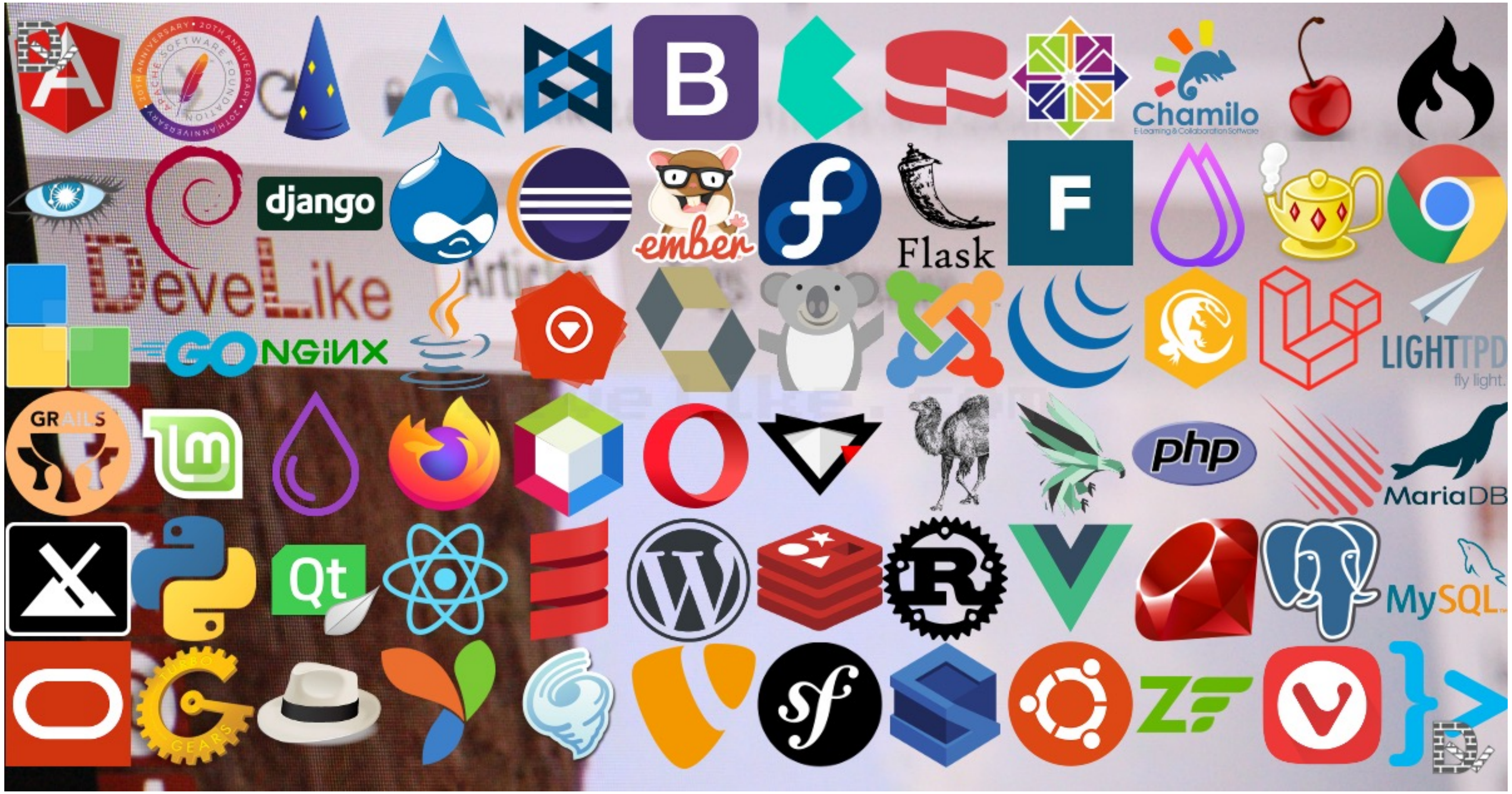
Software vs. program

- **Software**
  - **A collection or set of programs**, procedures, data, or instructions
  - They instruct computer about what to do
  - They are designed to perform well defined functions
- **Program**
  - **A collection of instructions** or ordered operations for computer
  - They perform specific function or perform a particular task
  - They achieve a specific result

Software vs. program

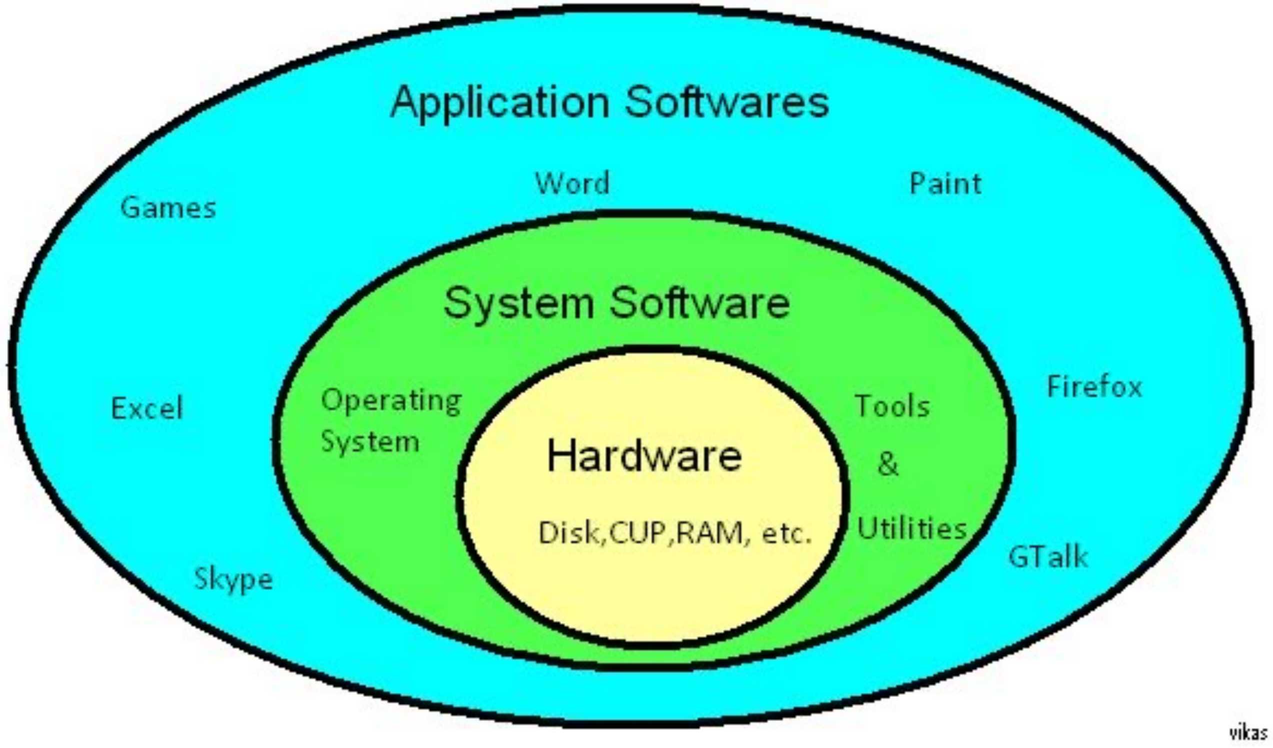
Software	Program
Various categories of software includes application software, system software, computer programming tools, etc	There are **no such categories of program**
Size of software generally ranges **from megabytes (Mb) to gigabytes (Gb)**	**Size** of program generally ranges **from kilobytes (Kb) to megabytes (Mb)**
Software's are usually **developed by** people having expert knowledge and experience as well as are trained in developing software and are also referred to as **software developers**	**Programs** are usually **developed by** person who is **beginner** and have **no prior experience**
It **requires more time to create software than program**	It **requires less time to create program than software**
…	…

Software


Chamilo
E-Learning & Collaboration Software
django
ember
Flask
F
DeveLike
GO
NGINX
LIGHTTPD
fly light.
GRAILS
php
MariaDB
Qt
MySQL
ZF

Software

- **Software can be divided into several types**
  - Based on its areas and implementations
  - Two main categories
    - System software
      Application software



System software

- **System software definition**
  - It allows the **user to interact with the hardware components** of a computer system
  - The **interface (**allows the users to interact with the core system**) is provided** by the software
  - The system software can be called **the main or the alpha software** of a computer system
  - It can be further divided into several major types

System software

- **System software examples**
  - **The operating system**
    - Governs and maintains the inter-cooperation of the components of a computer system
  - **The language processor**
    - Provides the methods for human-machine interactions with multiple languages
    - Divided into multiple types
      - Compiler
        - Convert High-Level Language into machine level language in one go
        - C, C++ and Scala use compilers
      - Interpreter
        - Convert High-Level Language into machine level language line-by-line
        - Python, Ruby and Java use an interpreter

System software

- **System software examples**
  - **Utility software**
    - The most basic type of software
    - Helping the management of the system
    - Examples
      - Antivirus Software
        - provide protection to the computer system from unwanted malware and viruses
        - V3, QuickHeal, McAfee etc.
      - Text-editors
        - Help the users to take regular notes and create basic text files
        - Notepad, Gedit, Emacs, Vim, etc.

System software

- **System software examples**
  - **Device drivers**
    - Acts as an interface between the various Input-Output devices and the users or the operating system
    - For example, printers, and web cameras come with a driver disk that is needed to be installed into the system to make the device run in the system
  - **BIOS**
    - Basic Input Output System
    - It is a small firmware that controls the peripheral or the input-output devices attached to the system
    - Also responsible for starting the OS or initiating the booting process

Application software

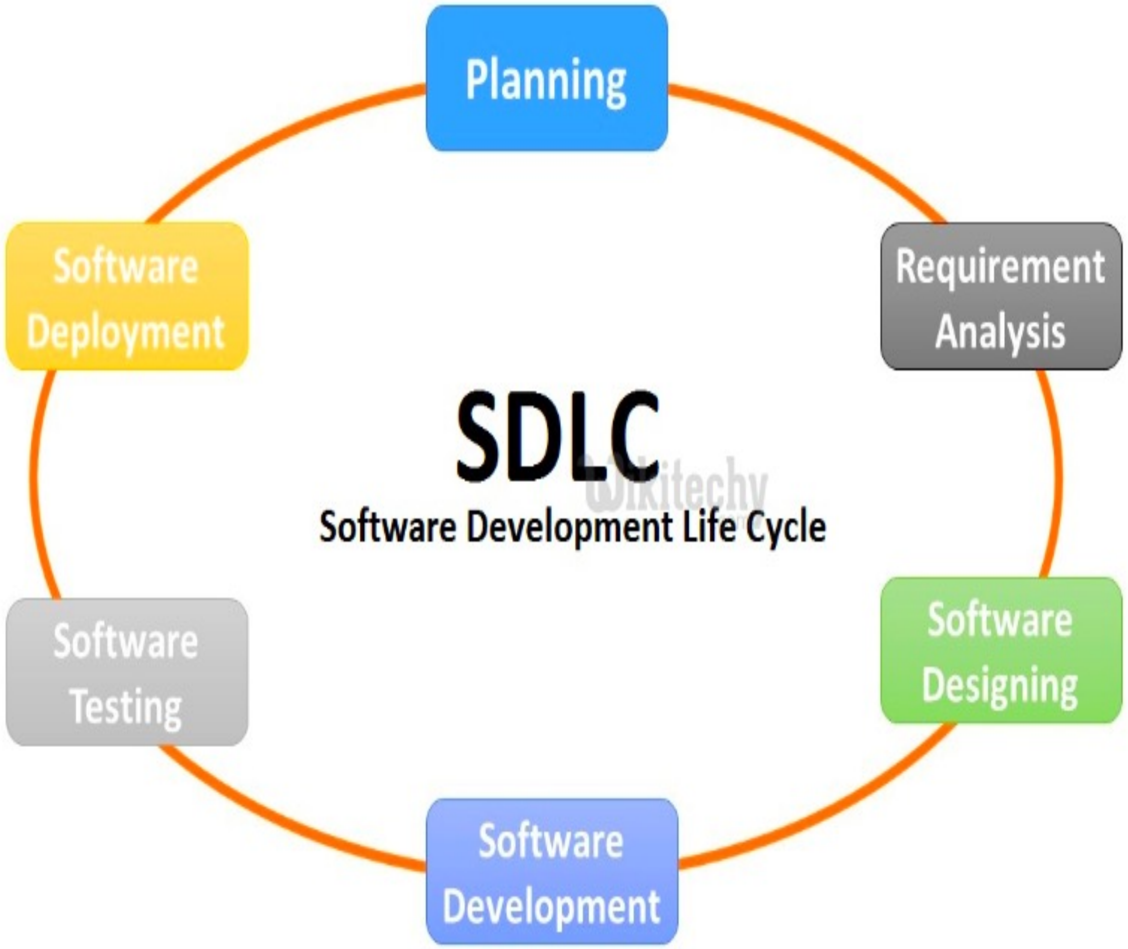
- **Application software**
  - **Used to run to accomplish a particular action and task**
  - Dedicated to performing simple and single tasks
    - E.g., a single software cannot serve both the reservation system and banking system
  - **Divided into two types**
    - **General purpose application software**: comes in-built and ready to use, manufactured by some company or someone
      - Microsoft Excel – Used to prepare excel sheets
      - VLC Media Player – Used to play audio/video files
      - Adobe Photoshop – Used for designing and animation and many more
    - **Specific purpose application software**: Is customizable and mostly used in real-time or business environments
      - Ticket Reservation System
      - Hotel Management System

Software development life cycle (SDLC)

Software development life cycle (SDLC)

- **Software development life cycle (SDLC)**
  - SDLC is a **common term of an efficient approach** for the software development
    - A process used in the software industry to produce software with the highest quality and lowest cost in the shortest time
  - The entire software development process includes **6 stages**
  - It **specifies the task(s)** to be performed at various stages by a software engineer/developer
  - It's **vital for a software developer** to **have prior knowledge** of this software development process
  - Google also educates the similar thing with this during the orientation

Software development life cycle (SDLC)



1. Planning phase

- **Software development life cycle (SDLC) - 1. Planning phase**
  - **Input**: Client proposal or requirements
  - **Requirement analysis** is the most important and fundamental phase in SDLC
  - Senior members (with other team members) perform the following things to define the various technical approaches to implement the project successfully with minimum risks
    - The basic project approach
    - The product feasibility study in the economical, operational and technical areas
    - Quality assurance requirements
    - Risk identification associated with the project
  - **Output**: Technical Feasibility Study/Project initiation

2. Analysis phase

- **Software development life cycle (SDLC) - 2. Analysis phase**
  - **Input**: Technical feasibility study and funding
  - Clearly define and document the **product requirements** and get them approved from the customer or the market analysts
  - **Output**
    - Software Requirement Specification (SRS)
    - Consists of all the product requirements to be designed and developed during the project life cycle

Let's take look at the IEEE SRS template

(Google also uses the similar form when we specify software requirements)

3. Design phase

- **Software development life cycle (SDLC) - 3. Design phase**
  - **Input**: SRS
  - Write a **Design Document Specification (DDS)**
    - Contains more than one design approach for the product architecture
    - Defines all the architectural modules of the product along with communications and data flows of sub modules and third parties
    - Be reviewed by all the important stakeholders and based on various parameter (e.g., risk assessment, design modularity, etc)
    - The internal design of all the modules should be clearly defined
  - Write **high level design (HLD)** / **detail design (DDD)** based on **DDS**
  - **Output**: DDS, HLD, and/or DDD

4. Build phase (implementation)

- **Software development life cycle (SDLC) - 4. Build phase**
  - **Input**: DDD (Detail Design Document)
  - The actual development starts and the product is built based on DDD
    ➔ *If the design is performed in a detailed and organized manner, code generation can be accomplished without much hassle*
  - Developers must follow the coding guidelines (e.g., Google C++ style guide)
  - **Output**: Program/software

5. Test phase

- **Software development life cycle (SDLC) - 5. Test phase**
  - **Input**: Un-tested program/software, Test plan document
  - Perform tests based on SRS
    - Testing only stage where product defects are reported, tracked, fixed and retested
    - Continuing it until the result reaches the quality standards defined in the SRS
  - **Output**: tested program/software

6. Deployment phase

- **Software development life cycle (SDLC) - 6. Deployment phase**
  - **Input**: Tested program/software, migration plan
  - Released formally in the appropriate market as per the business strategy of that organization
  - The product may first be released in a limited segment and tested in the real business environment (**UAT- User acceptance testing**)
  - After the product is released in the market, its maintenance is done for the existing customer base
  - **Output**: Tested program/software migrated into production and celebrate

Day in the life of a software engineer

Day in the life of a software engineer – general things to do

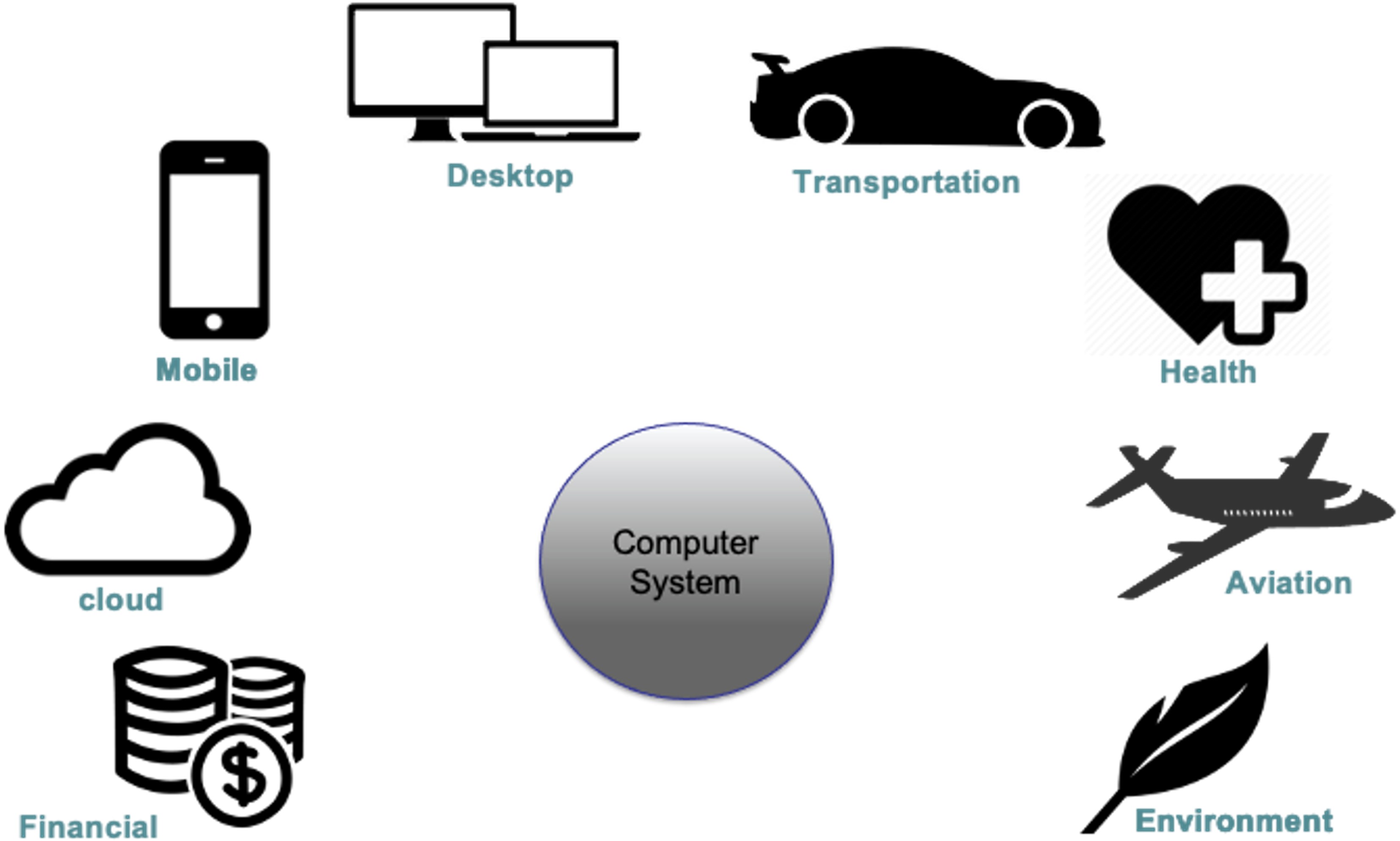
- **Software engineer works**
  - Set the goal with the manager for the upcoming 3 or 6 months
  - Work with a team, but individually contribute to the software
  - Sync with coworkers by team meetings, emails, and chat
  - Attend several in-company classes
    - E.g., language self-study, libraries, and tools
  - Attend bigger size meetings
    - Town hole meeting
    - All hands
  - Contributions to the community
    - Work as an interviewer
    - Learn and share new technologies
    - Voluntary works for outside communities

Day in the life of a software engineer – coding related things

- **Software engineer works**
  - Become familiar with team projects and codebases
  - Find out topics that we can contribute to
  - Write proposals or list up items
  - Share with other members and get feedback from others
  - Coding / testing / reviewing / submitting
  - Present what we have done with our team and others outside of our team

Two important skills for software engineers

Software failure



Desktop

Transportation

Mobile

Health

Computer System

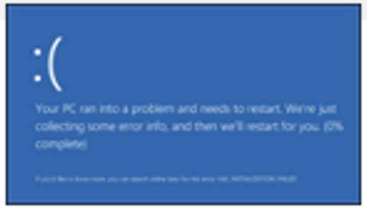
cloud

Aviation

Financial

Environment

Software failure



:(

Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you. (0% complete)

If you'd like to know more, you can search online later for this error: HAL INITIALIZATION FAILED

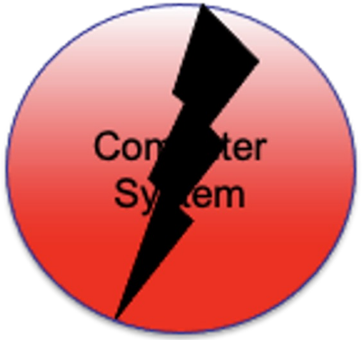
Crash

Accident

Mobile

Life

cloud



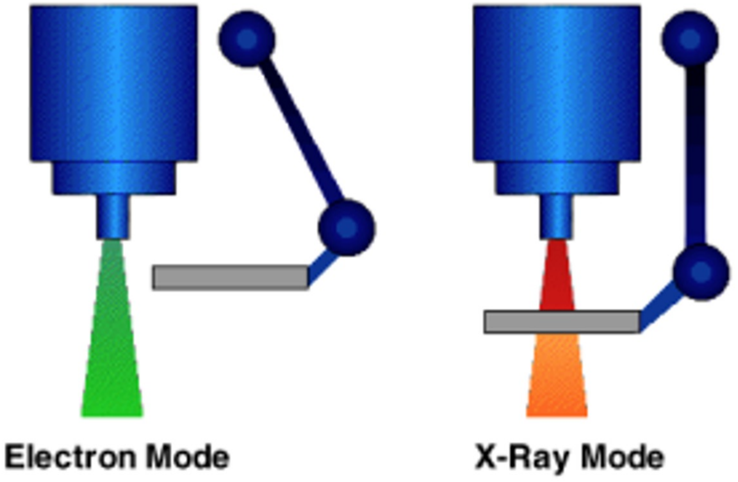
Loss

Financial

Environment

Software failure

- **Software failure examples**
  - **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 burn
  - Software race condition
    - Poor software design and QA
    - Misleading user interface
    - Root cause: Poor understanding of software reliability issue



Software failure

- **Software crisis examples**
  - **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

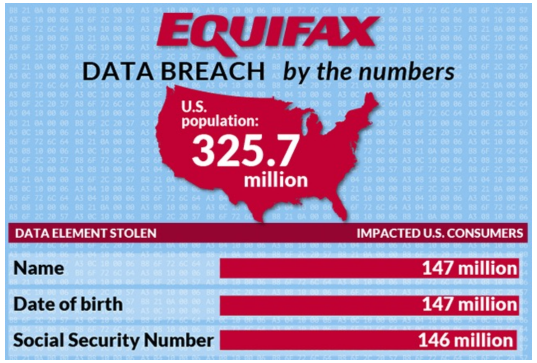
Software failure

Ariane 5 explosion
$370 million

1996

…

50% of American p
ersonal record



EQUIFAX

DATA BREACH *by the numbers*

U.S. population: 325.7 million

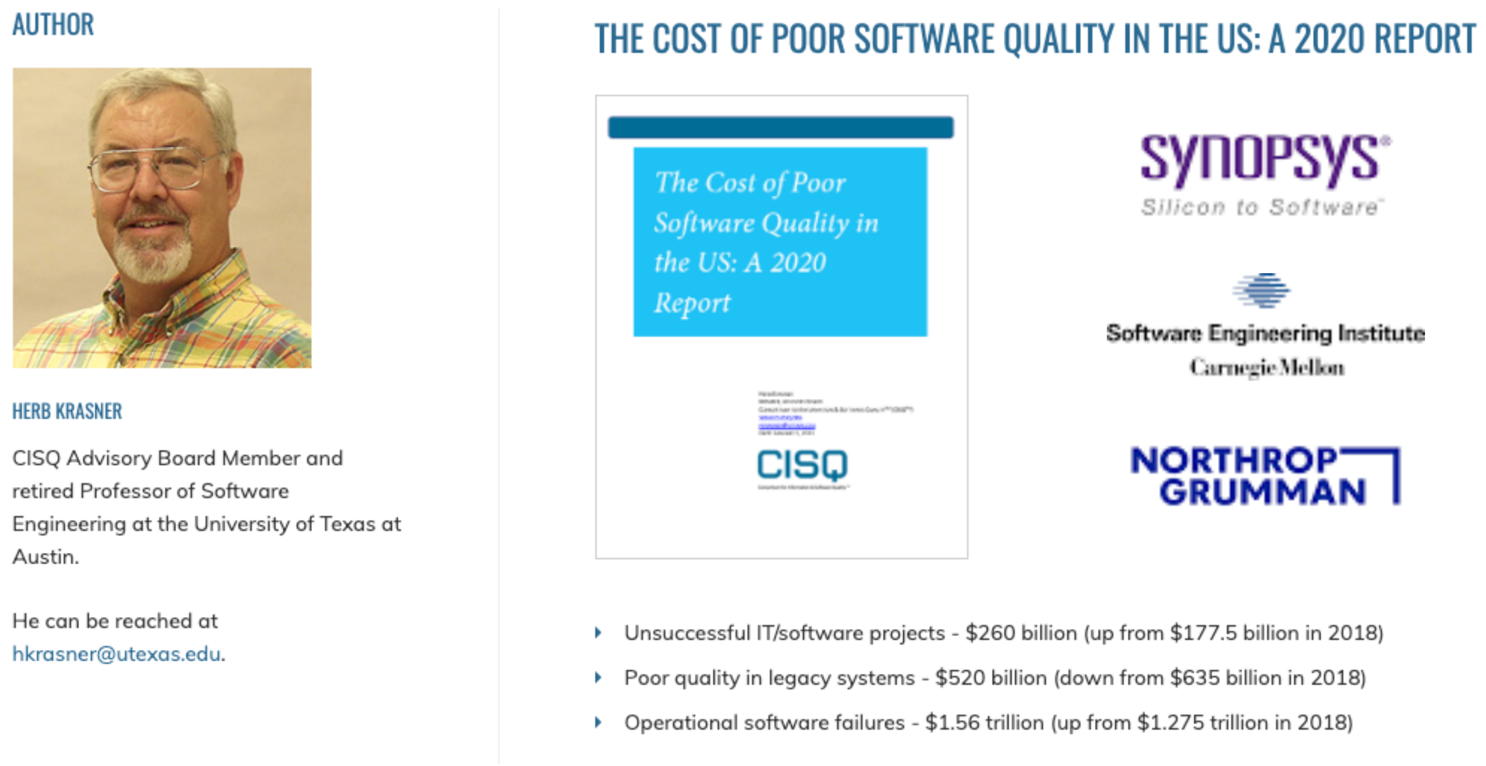
DATA ELEMENT STOLEN	IMPACTED U.S. CONSUMERS
Name	147 million
Date of birth	147 million
Social Security Number	146 million

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.

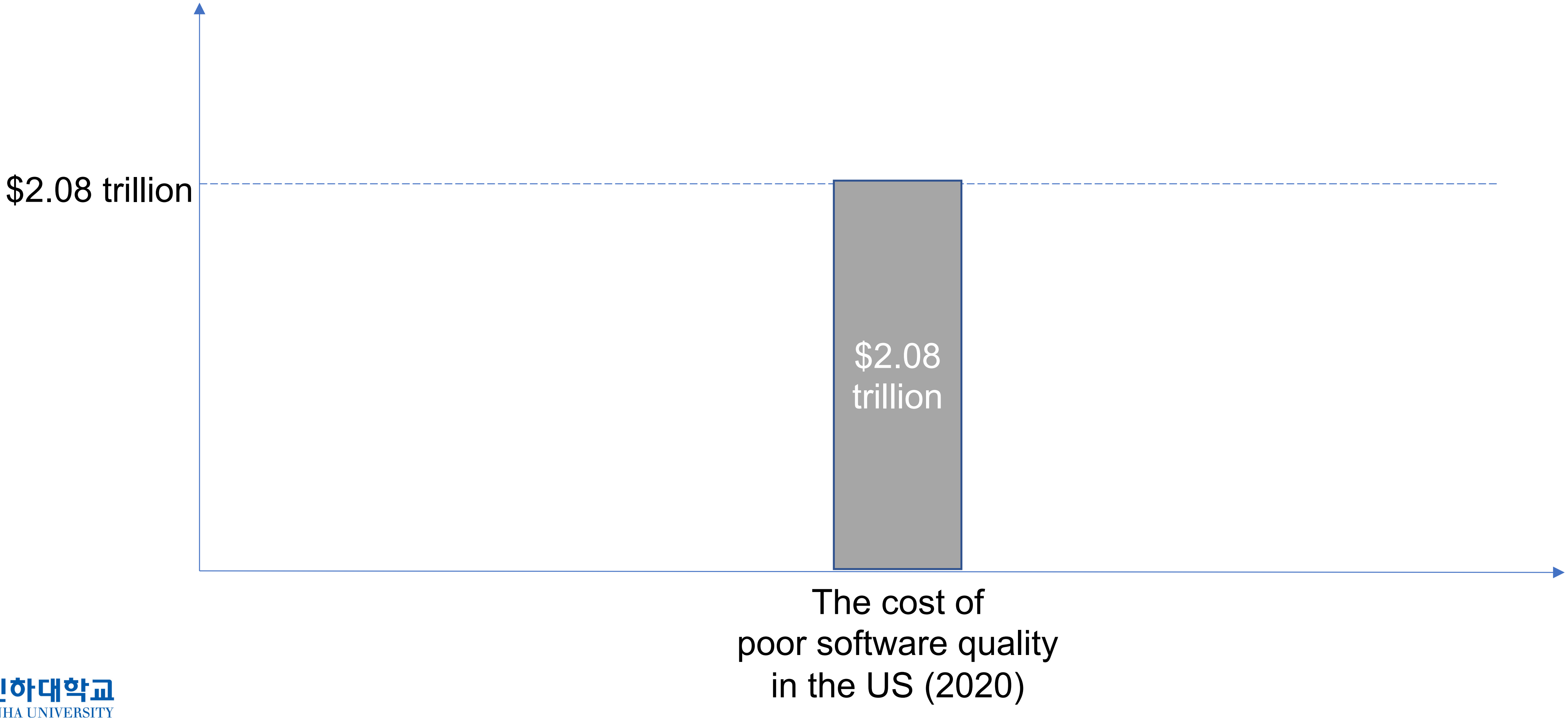
He can be reached at hkrasner@utexas.edu.

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

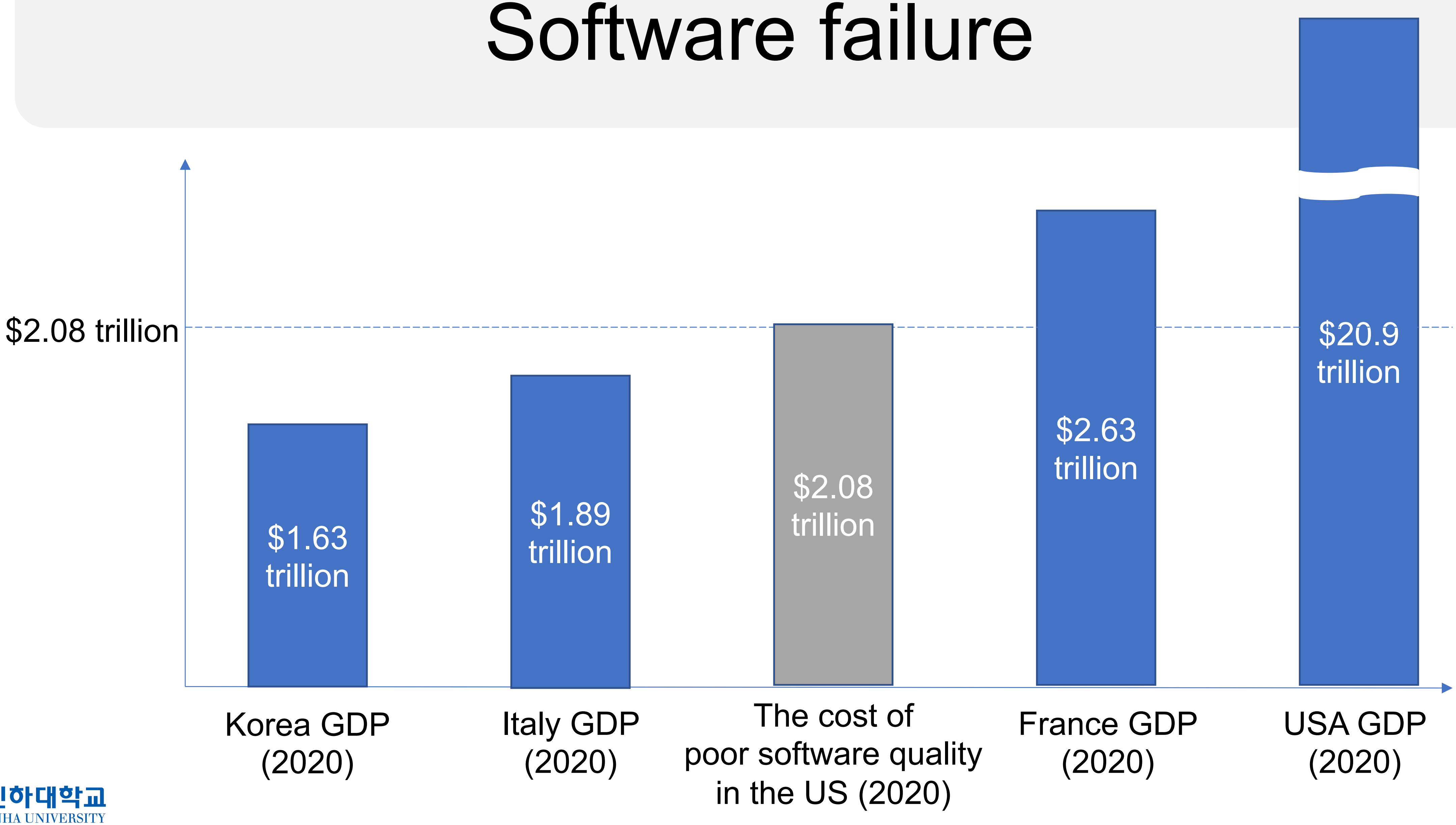
The Cost of Poor Software Quality in the US: A 2020 Report
CISQ
synopsys
Silicon to Software
Software Engineering Institute
Carnegie Mellon
NORTHROP GRUMMAN

- 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



인하대학교
INHA UNIVERSITY

Software failure

practical

	Expressiveness level	Assurance level	Cost level
Code review	Very high	Very low	Medium
Testing	Medium	Low	Medium
Type checker (Java, Haskell, Rust)	Low	High	low
Static alaysis (Coverity, Infer)	Medium	Medium	low
Formal verificaiton (UPPAAL, Z3, Adga, Coq)	Medium ~ High	High ~ Very high	Medium ~ Very high

Code review

Code review

- **Code review**
  - Careful, systematic study of source code by people who are not the original author of the code
  - Analogous to proofreading a term paper

Code review

- **Purpose of code review**
  - Can catch many bugs, design flaws early
  - > 1 person has seen every piece of code
    - Insurance against author's disappearance
    - Accountability (both author and reviewers are accountable)
  - Forcing function for documentation and code improvements
    - Authors to articulate their decisions
    - Authors participate in the discovery of flaws
    - Prospect of someone reviewing your code raises quality threshold
  - Inexperienced personnel get hands-on experience without hurting code quality
    - Pairing them up with experienced developers
    - Can learn by being a reviewer as well
      - Google has a readability reviewer class for Google software engineers

Code review

- **Purpose of code review - by numbers**
  - **From Steve McConnel's Code Complete**
  - Average defect detection rates
    - Unit testing: 25%
    - Function testing: 35%
    - Integration testing: 45%
    - Design and code inspections: 55% and 60%
  - 11 programs developed by the same group of people
    - First 5 without reviews: average 4.5 errors per 100 lines of code
    - Remaining 6 with reviews: average 0.82 errors per 100 lines of code
    - Errors reduced by > 80%
  - After AT&T introduced reviews, 14% increase in productivity and a 90% decrease in defects

Code review

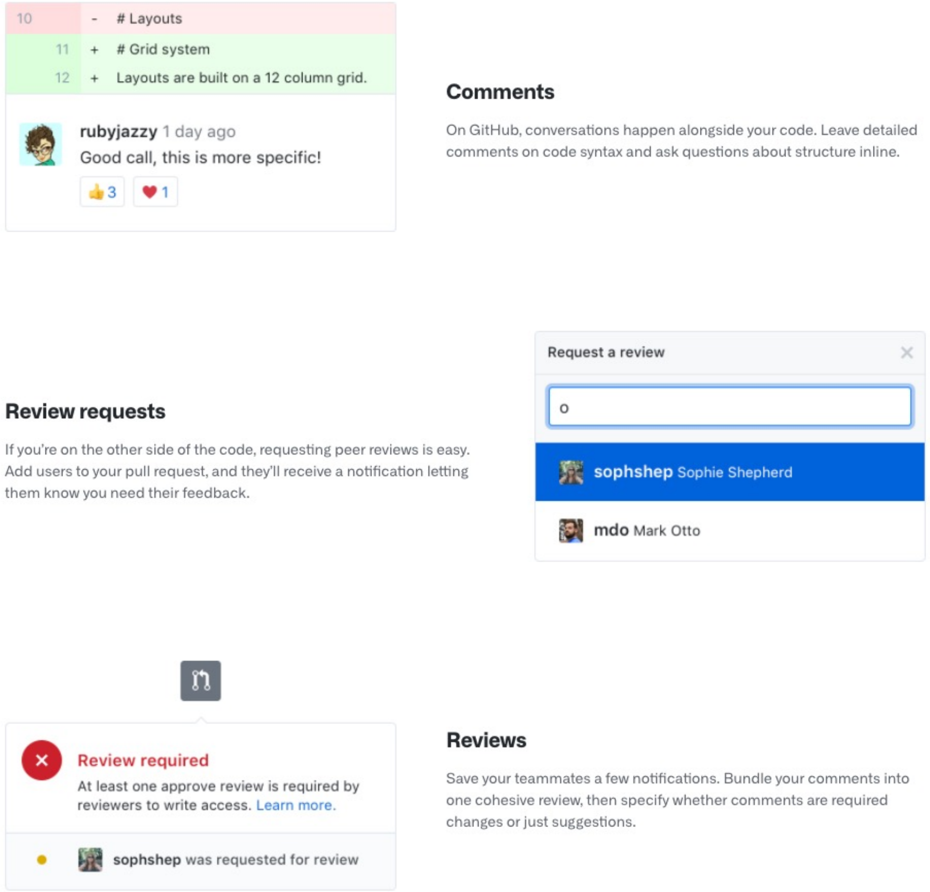
- **Code reviews should look at the following things**
  - **Design** - Is the code well-designed and appropriate for your system?
  - **Functionality** - Does the code behave as the author likely intended?
  - **Complexity** - Could the code be made simpler?
  - **Tests** - Does the code have correct and well-designed automated tests?
  - **Naming** - Did the developer choose clear names for variables, classes, methods, etc.?
  - **Comments** - Are the comments clear and useful?
  - **Style** - Does the code follow our style guides?
  - **Documentation** - Did the developer also update relevant documentation?
- Again, each project or company has its own guideline that developers have to follow (e.g., Google C++ style guide)

Code review

- **Code review in industry**
  - Code reviews are a very common industry practice
  - Made easier by advanced tools that
    - Integrate with configuration management systems
    - Highlight changes (i.e., diff function)
    - Allow traversing back into history
  - Google also provide integrated code review tool with our development tools.
  - To taste it, you can use [code review system of Github](#).

Code review

- **Code review in Github**



```
- # Layouts
+ # Grid system
+ Layouts are built on a 12 column grid.
```

rubyjazzy 1 day ago
Good call, this is more specific!
👍 3 ❤️ 1

Comments

On GitHub, conversations happen alongside your code. Leave detailed comments on code syntax and ask questions about structure inline.

Review requests

If you're on the other side of the code, requesting peer reviews is easy. Add users to your pull request, and they'll receive a notification letting them know you need their feedback.

Request a review ✕

o

sophshep Sophie Shepherd

mdo Mark Otto

✕ Review required

At least one approve review is required by reviewers to write access. Learn more.

● sophshep was requested for review

Reviews

Save your teammates a few notifications. Bundle your comments into one cohesive review, then specify whether comments are required changes or just suggestions.

Code review

- **How to do code review**
  - **No official style guide**
  - Strictly **enforcing personal style** in larger projects is **prohibited**
  - There are some rules that are quite sensible, **removing “bad smells”**
  - “Bad smells”
    - The coding patterns that lack expandability/flexibility and hinder future code development
    - The following things are frequent and easy to get rid of
      - **Duplicated code** is a maintenance nightmare
        - Extract it into a function that you can call where needed
      - **Magic numbers and strings** don’t tell the reader what they mean
        - Use named constants instead
      - **Dead code** makes your code complex and **redundant comments** hurt readability
        - Just throw it away

Code review

- **Common guidelines in code review**
  - The rule of the three: If a class defines one (or more) of the following, it should explicitly define all three, which are 1) destructor, 2) copy constructor, 3) copy assignment operator
  - Do not use #define unless you have to use it
  - Try to use **const** member functions and variables
  - Set up the criteria on class, function, field, and variable names
  - Locate functions in proper classes
  - Try to use initializer list
  - Use iteration over STL containers
  - …

```
#define Fresh 1
#define Sophomore 2
#define Junior 3
#define Senior 4

class Student {
  public:
  Student(int id, int year) {
    student_id = id;
    student_year = year;
  };

  ~Student();

  int GetStudentID() { return student_id; }
  int get_student_year() { return student_year; }

  private:
  int student_id;
  int student_year;
};


bool FindStudent(int id, std::vector<Student> students) {
  for (int i = 0; i < students.size(); i++) {
    if (students[i].GetStudentID() == id) {
      return true;
    }
  }
  return false;
}
```

```
#define Fresh 1
#define Sophomore 2
#define Junior 3
#define Senior 4

class Student {
  public:
  Student(int id, int year) {
    student_id = id;
    student_year = year;
  };

  ~Student();

  int GetStudentID() { return student_id; }
  int get_student_year() { return student_year; }

  private:
  int student_id;
  int student_year;
};


bool FindStudent(int id, std::vector<Student> students) {
  for (int i = 0; i < students.size(); i++) {
    if (students[i].GetStudentID() == id) {
      return true;
    }
  }
  return false;
}
```

Do not use #define

Violate the rule of the three

Initializer list is not used

Inconsistency in function names

Fields are not distinguishable from local variables

References should be used

Iterator is not used

```
class Student {
  public:
  enum StudentYear { FRESH = 1, Sophomore, Junior, Senior };

  Student(const int id, const StudentYear year) :
    id_(id), year_(year) {};
  Student(const Student& student) :
    id_(student.id_), year_(student.year_) {};
  Student& operator=(const Student& student) {
    if (this != &student) {
      *this = Student(student);
    }
    return *this;
  };
  ~Student();

  int GetId() { return id_; }
  int GetYear() { return year_; }

  bool FindStudent(const int id,
    const std::vector<Student>& students) const {
    for (const auto& student : students) {
      if (student.id_ == id) {
        return true;
      }
    }
    return false;
  }

  private:
  const int id_;
  StudentYear year_;
};
```

```
int STANDARD=0, BUDGET=1, PREMIUM=2, PREMIUM_PLUS=3;

class Account {
  public:
  double principal,rate; int daysActive,accountType;
};

double calculateFee(std::vector<Account> accounts)
{
  double totalFee = 0.0;
  Account account;
  for (int i=0;i<accounts.size();i++) {
    account=accounts[i];
    if ( account.accountType == PREMIUM ||
      account.accountType == PREMIUM_PLUS )
      totalFee += .0125 * (             // 1.25% broker's fee
        account.principal * pow(account.rate,
        (account.daysActive/365.25))
        - account.principal);   // interest-principal
  }
  return totalFee;
}
```

Software testing & unit testing

Software testing

- **Software testing**
    - Evaluation of the software against requirements gathered from 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
def numZero (arr):
   count = 0
   for i in range(1, len(arr)):
     if arr[i] == 0:
        count += 1

   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
```

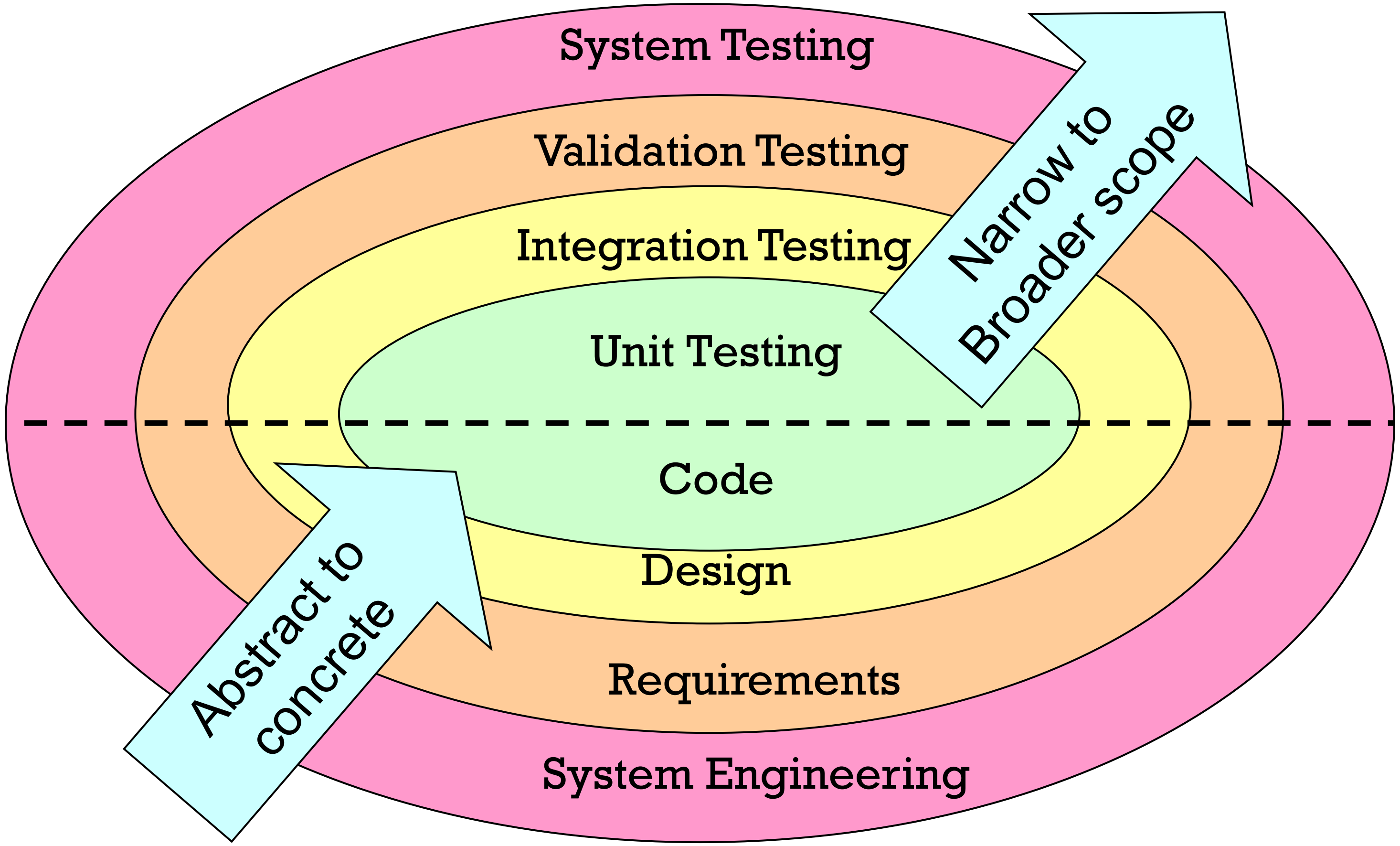
Error: i is 1, not 0
Error propagates to the variable count
Failure: count is 0 at the return statement

Software testing

- **Does testing works?**
  - "measuring over 20 projects: if you have a large number of unit 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

Software testing

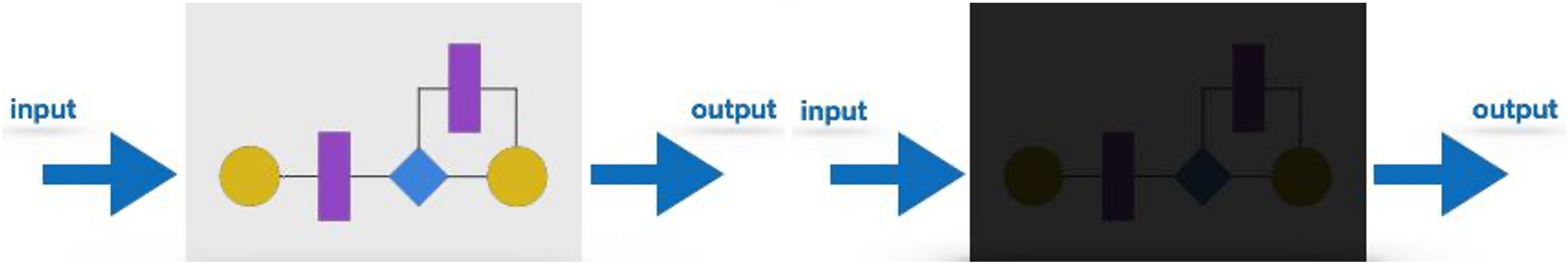


Software testing

- **Testing – levels**
  - **Unit testing**
    - Concentrates on each component/function of the software as implemented in the code
    - Exercises specific paths in a component's control structure to ensure complete coverage and maximum error detection
  - **Integration testing**
    - Focuses on the design and construction of the software architecture
    - Focuses on inputs and outputs, and how well the components fit and work together
  - **Validation testing**
    - Requirements are validated against the constructed software
    - Provides final assurance that the software meets all functional, behavioral, and performance requirements
  - **System testing**
    - The software and other system elements are tested as a whole
    - Verifies that all system elements (software, hardware, people, databases) mesh properly and that overall system function and performance is achieved

Software testing

- **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.



인하대학교
INHA UNIVERSITY

Software testing

- **Testing – types, tactics, and techniques**
  - **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, configuration testing, compatibility testing, regression testing, maintenance testing, usability testing, etc.

Software testing

- **Unit testing frameworks and libraries**
  - **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**
  - 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
  - **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

Create tests

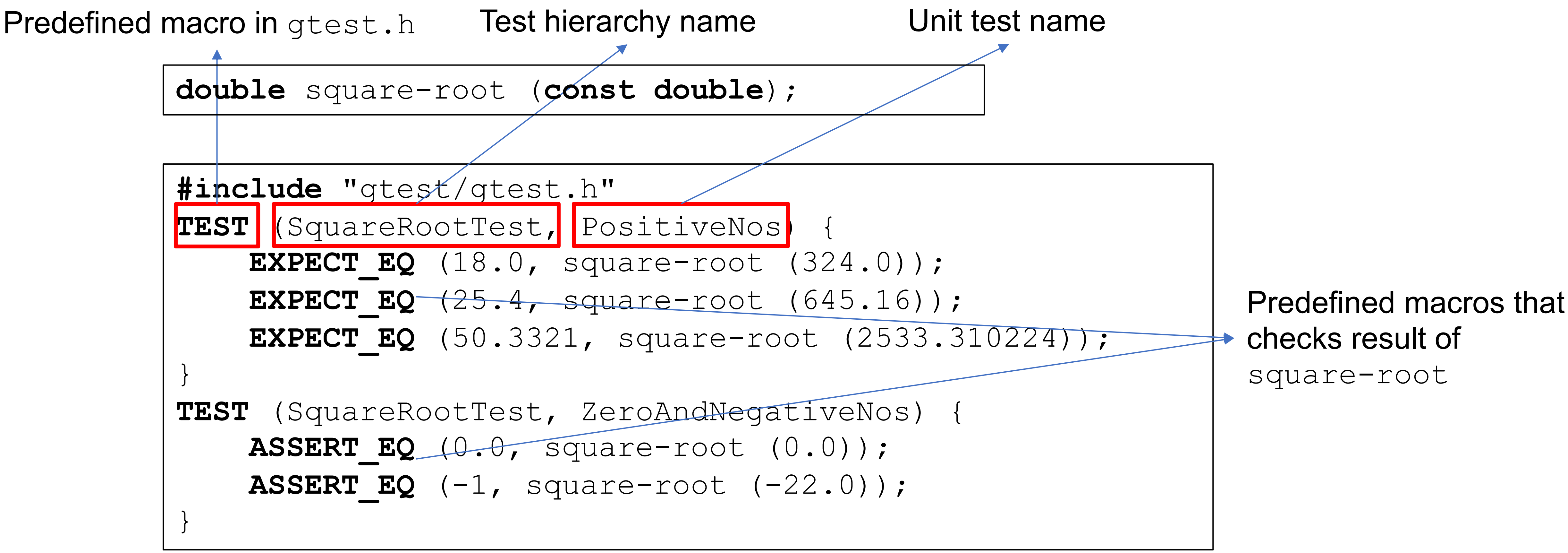
- **Creating a basic test**
  - **Target code: prototype for square-root**

```
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));
}
```

Create tests



Predefined macro in `gtest.h`

Test hierarchy name

Unit test name

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

```
#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 `square-root`

Check results

- **Assertions in Googletest**
  - Google Test assertions are macros that resemble function calls
  - You test a class or function by making assertions about its behavior
  - `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 function
    - Test will directly fail if the assertion is not satisfied

Check results

- Basic assertions

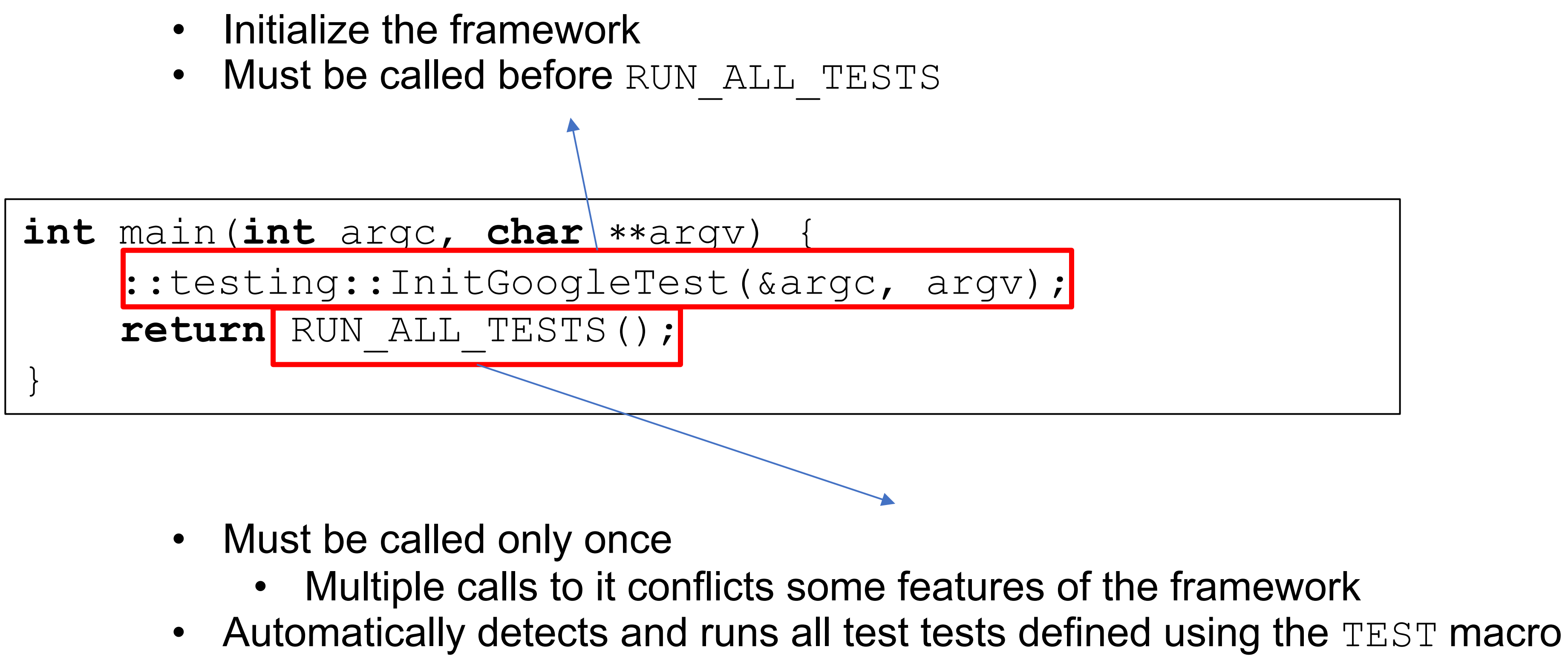
Fatal assertion	Nonfatal assertion	Verifies
`ASSERT_TRUE(condtion);`	`EXPECT_TRUE(condtion);`	`Condition` **is true**
`ASSERT_FALSE(condition);`	`EXPECT_FALSE(condition);`	`Condition` **is false**

Check results

- 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

Run tests



- 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

Run tests

```
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
```

Python unittest

- A module in the Python standard library that provides various automations 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

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

Create & run tests

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

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

Create & run tests

```
# 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()
```

Create & run tests

```
# 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
```

Create & run tests

```
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()
```

Check results

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)

Questions?

Code review example

```
// An individual account.  Also see XXX.
class Account {
  public:
  // Constructor with default value.
  // It is better for us to provide those default values with config files.
  Account() :
  broker_fee_percent_(0.0125),
  days_per_year_(365.25) {};

  // Constructor with user-defined broker fee percent and days per year.
  Account(double broker_fee_percent, double days_per_year) :
    broker_fee_percent_(broker_fee_percent),
    days_per_year_(days_per_year) {};

  // Copy constructor
  Account(const Account& account) :
  broker_fee_percent_(account.broker_fee_percent_),
    days_per_year_(account.days_per_year_),
    principal_(account.principal_),
    rate_(account.rate_),
    days_active_(account.days_active_),
    account_type_(account.account_type_) {};

  Account& operator=(const Account& account) {
    if (this != &account) {
      *this = Account(account);
    }
    return *this;
  };
  ~Account();
```

```
// The varieties of account our bank offers.
enum AccountType {STANDARD, BUDGET, PREMIUM, PREMIUM_PLUS};

// Compute interest.
double GetInterest() {
  double years = days_active_ / days_per_year_;
  double compound_interest = principal_ * pow(rate_, years);
  return compound_interest - principal_;
}

// Return true if this is a premium account.
bool IsPremium() {
  return account_type_ == AccountType::PREMIUM ||
    account_type_ == AccountType::PREMIUM_PLUS;
}

// Return the sum of the broker fees for all the given accounts.
double CalculateFee(std::vector<Account>& accounts) {
  double total_fee = 0.0;
  for (Account account : accounts) {
    if (IsPremium()) {
      total_fee += broker_fee_percent_ * GetInterest();
    }
  }
  return total_fee;
}
```

```
 private:
 // The portion of the interest that goes to the broker.
 const double broker_fee_percent_;
 // The number of days per one year.
 const double days_per_year_;
 double principal_;
 // The yearly, compounded rate (at 365.25 days per year).
 double rate_;
 // Days since last interest payout.
 int days_active_;
 // Account type.
 AccountType account_type_;
};
```