

CTIS359

Principles of Software Engineering

Introduction to SW Testing

***“Testing shows the presence, not
the absence of bugs.”***

Edsger W. Dijkstra

Today

- Why do we need testing? Why it is important?
- Lots of definitions related to software testing
 - Testing, Validation & Verification (V&V)
- Software errors, faults and failures
 - Differences / Different definitions
- 9 causes of software errors
- Categorizations
 - Static vs. Dynamic
 - White-Box vs. Black-Box
 - Functional vs. Non-functional

Why do we need testing?

Testing...

- One of the unfortunate facts about bugs is that you can never tell when they're ALL gone.
- "Testing shows the presence, not the absence of bugs."
- You can run tests as long as you like, but **you can never be sure you've found every bug.**

Why do we need testing?

Testing...

- One can say that ...
- "As we'll never be sure that our software is bug-free, let's skip testing..."
- You can never be sure that your software is bug-free.

Why Important?

- Testing is important, to make sure that
 - All **business requirements** are implemented
 - **Functionalities** are behaving as expected
 - Application is **secure enough**
 - **Do not break** under working circumstances
 - **Performs** its functions within an acceptable time
 - Works well on all supported **OS, devices**
 - To make sure that **customer** can trust and rely on your project

Why is Testing (a separate activity in SDLC) required?

- Testing provides an assurance to the stakeholders that product works as intended.
- Avoidable defects leaked to the end user/customer without proper testing adds **bad reputation** to the development company.
- Separate testing phase adds a layer of confidence to the stakeholders regarding quality of the product developed.
- Testing team adds another dimension to the product development by providing a check on the product development process.



Why is Testing (a separate activity in SDLC) required?

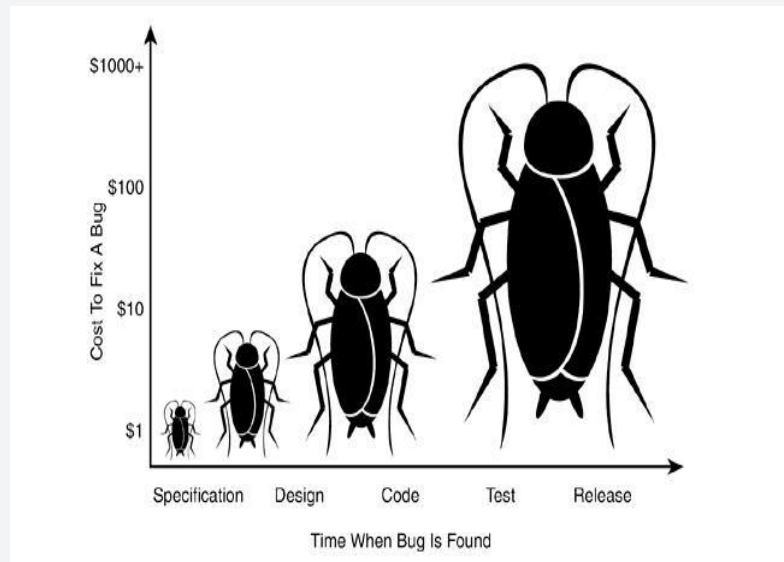
- Testing provides an assurance to the stakeholders that product works as intended.
- Avoidable defects leaked to the end user/customer without proper testing adds **bad reputation** to the development company.
- Separate testing phase adds a **confidence factor** to the stakeholders regarding **quality of the software** developed.
- Testing team adds another dimension to the software development by providing a different view point to the product development process.

Why is Testing (a separate activity in SDLC) required?



- Provides assurance to the stakeholders that the software meets the requirements.
- Prevents the software from reaching the end user/customer, which would result in **bad reputation** to the organization.
- Builds a **confidence factor** to the **quality of the software**.
- Testing team **adds another dimension** to the software development by providing **a different view point** to the product development process.

Software Testing Importance



- Defects detected in earlier phase of SDLC results into **lesser cost** and **resource utilization** for defect resolution.
- Saves development **time** by detecting issues in earlier phase of development.



The Goal of Software Testing

- The goal of testing is to **uncover as many defects**, at the highest level of seriousness, **as possible**.
- It is **NOT** possible to test an application with **every possible input value** due to the extraordinarily **large # of combinations** of input values.
 - For this reason, **testing can detect the presence of defects but not their absence**.



Software Testing

- A false statement: *"It has been thoroughly tested and therefore has no defects."* !!!!
- Thorough testing is nevertheless indispensable.

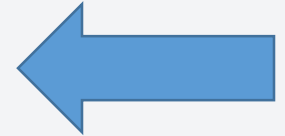


7 principles in software testing

- Testing shows presence of defects
- Exhaustive testing is not possible
- Early testing
- Defect clustering (kümelenmesi)
- Pesticide paradox
- Testing is context dependent
- Absence of errors fallacy (safsata)
 - **Sources:**
 - <https://www.boxuk.com/insight/the-seven-principles-of-testing/>
 - <https://www.softwaretestinghelp.com/7-principles-of-software-testing/>

7 principles in software testing

- **Testing shows presence of defects**
- Exhaustive testing is not possible/infeasible
- Early testing
- Defect clustering (kümelenmesi)
- Pesticide paradox
- Testing is context dependent
- Absence of errors fallacy (safsata)




Exhaustive testing is not possible/infeasible

- Space is generally too big to cover exhaustively
- Imagine exhaustively testing a **32-bit floating-point** multiply operation, $a*b$
 - There are 2^{64} test cases!
 - 18,446,744,073,709,551,616 test cases!

Statistical testing doesn't work for software

- Other engineering disciplines can test small random samples (e.g. 1% of hard drives manufactured) and infer defect rate for whole lot.
- Many tricks (Ikea, Arçelik) to speed up time (e.g. opening a refrigerator 1000 times in 24 hours instead of 10 years)
- Gives known failure rates (e.g. mean lifetime of a HDD)
- But assumes continuity or uniformity across the space of defects, which is true for physical artifacts.
- **This is not true** for software
 - overflow bugs (Ariane 5) happen abruptly
 - Pentium division bug affected approximately 1 in 9 billion divisions


7 principles in software testing

- Testing shows presence of defects
- Exhaustive testing is not possible
- Early testing 
- Defect clustering (kümelenmesi)
- Pesticide paradox
- Testing is context dependent
- Absence of errors fallacy (safsata)

Early Testing

- To find the defect in the software, early test activity shall be started. The defect detected in the early phases of SDLC will be very less expensive. For better performance of software, software testing will start at the initial phase i.e. testing will perform at the requirement analysis phase.
- **Involving testing early is also a fundamental Agile principle**, which sees testing as an activity throughout, rather than a phase (which in a traditional waterfall approach would be at the end) because it enables quick and timely continuous feedback loops. When a team encounters hurdles or impediments, early feedback is one of the best ways to overcome these, and testers are essential for this.
- Consider the tester as the '**information provider**' – a valuable role to play.

7 principles in software testing

- Testing shows presence of defects
- Exhaustive testing is not possible
- Early testing
- Defect clustering (kümeleşme) 
- Pesticide paradox
- Testing is context dependent
- Absence of errors fallacy (safsata)


Defect Clustering

- This is the idea that certain components or modules of software usually contain the most number of issues, or are responsible for most operational failures. Testing therefore, should be focused on these areas (proportionally to the expected – and later observed – defect density of these areas).
- **Pareto Principle** to software testing state that 80% of software defect comes from 20% of modules.

Defect Clustering

- Possible reasons for defect clustering:
 - This is particularly the case with large and complex systems
 - A more complicated component
 - Components with more third-party dependencies
 - Inheriting legacy code, and developing new features in certain components that are undergoing frequent changes and are therefore more volatile, can also cause defect clustering.

7 principles in software testing

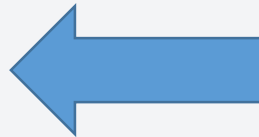
- Testing shows presence of defects
- Exhaustive testing is not possible
- Early testing
- Defect clustering (kümelenmesi)
- Pesticide paradox 
- Testing is context dependent
- Absence of errors fallacy (safsata)

Pesticide Paradox

- This is based on the theory that when you use pesticide repeatedly on crops, insects will eventually build up an immunity, rendering it ineffective. Similarly with testing, if the same tests are run continuously then – while they might confirm the software is working – eventually they will fail to find new issues.
- It is important to keep reviewing your tests and modifying or adding to your scenarios to help prevent the pesticide paradox from occurring – maybe using varying methods of testing techniques, methods and approaches in parallel.
 - If required a new set of test cases can be added and the **existing test cases can be deleted** if they are not able to find any more defects from the system.

7 principles in software testing

- Testing shows presence of defects
- Exhaustive testing is not possible
- Early testing
- Defect clustering (kümelenmesi)
- Pesticide paradox
- Testing is context dependent
- Absence of errors fallacy (safsata)



Testing is context dependent

- Testing is **ALL about the context**. The methods and types of testing carried out can completely depend on the context of the software or systems.
- **What you are testing will always affect your approach.**
- Different domains are tested differently, thus testing is purely based on the context of the domain or application.
- The risk associated with each type of application is different, thus it is not effective to use the same method, technique, and testing type to test all types of application.

7 principles in software testing

- Testing shows presence of defects
- Exhaustive testing is not possible
- Early testing
- Defect clustering (kümelenmesi)
- Pesticide paradox
- Testing is context dependent
- Absence of errors fallacy (safsata)



Absence of errors fallacy

- 99% of bug-free software may still be unusable, if wrong requirements were incorporated into the software and the software is not addressing the business needs.
- If your software or system is unusable (or does not fulfill users' wishes) then it does not matter how many defects are found and fixed – it is still unusable → it is irrelevant how issue- or error-free your system is; if the usability is poor users are unable to navigate, or/and it does not match business requirements then it has failed, despite having few bugs.
- It is important, therefore, to run tests that are relevant to the system's requirements.
- You should also be testing your software with users.
- Remember, just because there might be a low number of issues, it does not mean your software is shippable – meeting client expectations and requirements are just as important as ensuring quality.

Aims of Testing

- **What are we trying to do?**
 - find bugs as **cheaply** and **quickly** as possible
- **Reality vs. Ideal**
 - Ideally, choose one test case (TC) that exposes a bug and run it
 - In practice, have to run **many** TCs that “fail” (because they don’t expose any bugs)
- **In practice, conflicting desiderata**
 - increase chance of finding bug
 - decrease cost of test suite (cost to generate, cost to run)

Definitions

- **Testing:** Evaluating software by **observing its execution**.
- **Test Failure:** Execution that results in a failure.
- **Debugging:** The process of finding a fault given a failure.

The limitations of software testing

- The most important limitations of software testing is that testing can show **only the presence of failures, not their absence**.
- This is a fundamental, theoretical limitation; generally speaking, the problem of finding all failures in a program is undecidable.
- Testers often call a successful (or effective) test one that finds an error.

Software Testing



Türkçe - İngilizce

- Fault = bozukluk, bozulma
- Error = hata
- Failure = aksama, arıza
- Testing = sinama
- Validation = geçerleme
- Verification = doğrulama
- <http://bilisimde.ozenliturkce.org.tr>

Fault vs. Error vs. Failure

- **Software Fault:** A **static defect** in the software.
- **Software Error:** An incorrect **internal state** that is the manifestation of some **fault**.
- **Software Failure:** External, incorrect behavior **with respect to the requirements or other description of the expected behavior**.

Fault vs. Error vs. Failure

- Consider a medical doctor making a diagnosis for a patient.
 - The patient enters the doctor's office with a list of *failures* (that is, *symptoms*).
 - The doctor then must discover the *fault*, or *root cause of the symptom*.
 - To aid in the diagnosis, a doctor may order tests (such as high blood pressure, an irregular heartbeat, high levels of blood glucose, or high cholesterol) that look for anomalous internal conditions.
 - In our terminology, these *anomalous internal conditions* correspond to *errors*.
- Specifically, *faults* in *software* are *design mistakes*. They do not appear spontaneously, but rather exist as a result of some (unfortunate) decision by a human.

Fault vs. Error vs. Failure

```
public static int numZero (int[] x) {  
    // Effects: if x == null throw NullPointerException  
    // else return the number of occurrences of 0 in x  
    int count = 0;  
    for (int i = 1; i < x.length; i++){  
        if (x[i] == 0){  
            count++;  
        }  
    }  
    return count;  
}
```

Fault vs. Error vs. Failure

```
public static int numZero (int[] x) {  
    // Effects: if x == null throw NullPointerException  
    // else return the number of occurrences of 0 in x  
    int count = 0;  
    for (int i = 1; i < x.length; i++) {  
        if (x[i] == 0) {  
            count++;  
        }  
    }  
    return count;  
}
```

What is the fault?

Fault vs. Error vs. Failure

```
public static int numZero (int[] x) {  
    // Effects: if x == null throw NullPointerException  
    // else return the number of occurrences of 0 in x  
    int count = 0;  
    for (int i = 1; i < x.length; i++) {  
        if (x[i] == 0) {  
            count++;  
        }  
    }  
    return count;  
}
```

What is the fault?

The fault in this code is that it starts looking for 0s **at index 1** instead of index 0, as is necessary for arrays in Java.

Ex: `numZero ([2, 7, 0])` correctly evaluates to 1
`numZero ([0, 7, 2])` incorrectly evaluates to 0

In **both** of these cases **the fault** is executed.

Fault vs. Error vs. Failure

```
public static int numZero (int[] x) {  
    // Effects: if x == null throw NullPointerException  
    // else return the number of occurrences of 0 in x  
    int count = 0;  
    for (int i = 1; i < x.length; i++) {  
        if (x[i] == 0) {  
            count++;  
        }  
    }  
    return count;  
}
```

What is the failure?

Fault vs. Error vs. Failure

```
public static int numZero (int[] x) {  
    // Effects: if x == null throw NullPointerException  
    // else return the number of occurrences of 0 in x  
    int count = 0;  
    for (int i = 1; i < x.length; i++) {  
        if (x[i] == 0) {  
            count++;  
        }  
    }  
    return count;  
}
```

What is the failure?

Although both of these cases result in an **error**, only the second case results in **failure**.

Fault vs. Error vs. Failure

- To understand the error states, we need to identify the **state for the program**.
 - For the first example given above, the state at the if statement on the very 1st iteration of the loop is ($\mathbf{x} = [2, 7, 0]$, $\mathbf{count} = 0$, $\mathbf{i} = 1$, $\mathbf{PC} = \mathbf{if}$).
 - Notice that this state is in error precisely because the value of \mathbf{i} **should be 0 on the 1st iteration**.
 - However, since the value of \mathbf{count} is coincidentally correct, the error state does not propagate to the output, and hence the software does not fail.
 - In other words, a state is in error simply if it is not the expected state, even if all of the values in the state, considered in isolation, are acceptable.
 - More generally, if the required sequence of states is s_0, s_1, s_2, \dots , and the actual sequence of states is s_0, s_2, s_3, \dots , then state s_2 is in error in the second sequence.
 - In the second case the corresponding (error) state is ($\mathbf{x} = [0, 7, 2]$, $\mathbf{count} = 0$, $\mathbf{i} = 1$, $\mathbf{PC} = \mathbf{if}$).
 - In this case, the error propagates to the variable \mathbf{count} and is present in the return value of the method. Hence a failure results.


```
public static int numZero (int[] x) {  
    //Effects: if x==null throw NullPointerException  
    //          else return the number of occurrences of 0 in x  
    int count = 0;  
    for (int i = 1; i < x.length; i++) {  
        if (x[i]==0) {  
            count++;  
        }  
    }  
    return count;  
}
```

Error State:

x = [2,7,0]

i = 1

count = 0

PC=first iteration **for****Expected State:**

x = [2,7,0]

i = 0

count = 0

PC=first iteration **for****Fix:** for(int i=0; i<x.length; i++)x = [2,7,0], fault executed, **error**, no failurex = [0,7,2], fault executed, **error**, failure**State of the program:** x, i, count, PC

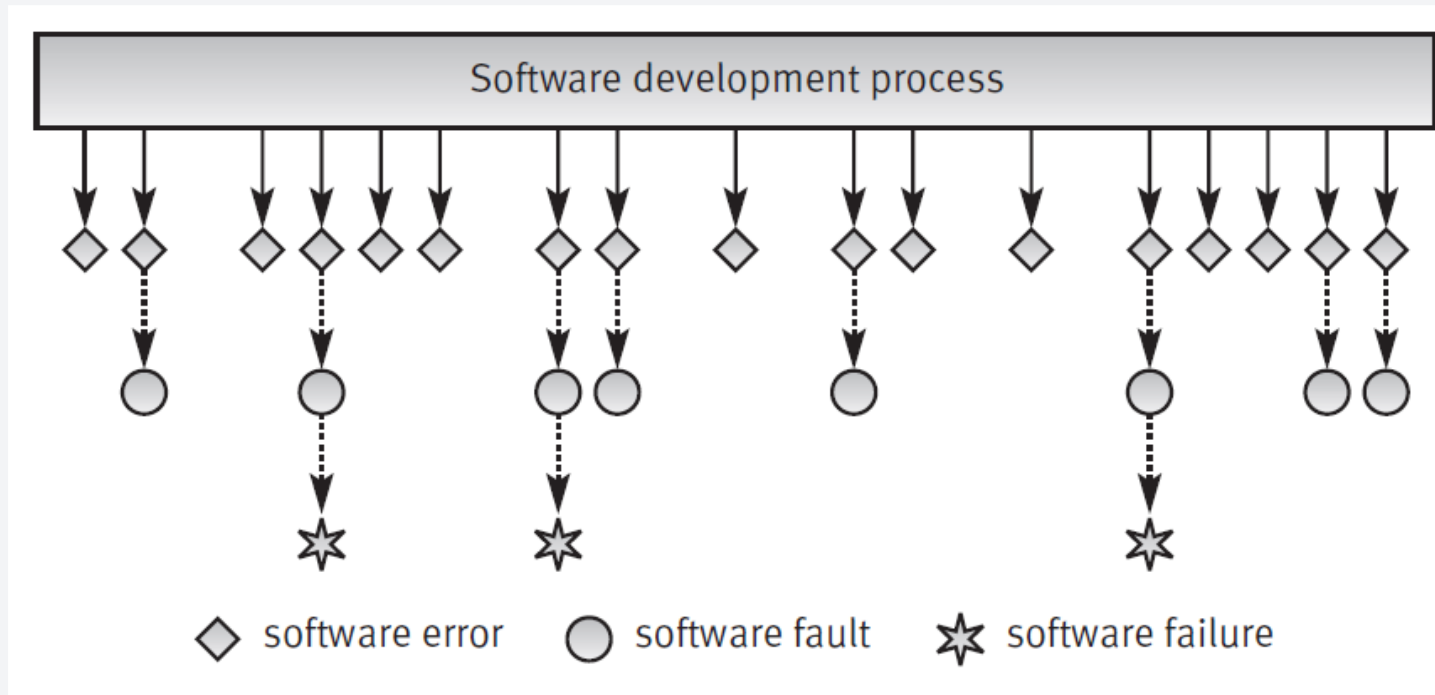
Fault vs. Error vs. Failure

Inconsistent
usage of terms

- **error**
- **1.** human action that produces an incorrect result
 - *[IEEE 1044-2009 IEEE Standard Classification for Software Anomalies, 2]*
- **2.** difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition
- **3.** erroneous state of the system
 - *[ISO/IEC 15026-1:2013 Systems and software engineering — Systems and software assurance — Part 1: Concepts and vocabulary, 3.4.4]*

Fault vs. Error vs. Failure

Error vs. Fault vs. Failure



In this figure, the development process yields 17 software errors, only 8 of which become software faults. Of these faults, only 3 turnout to be software failures.

The nine causes of software errors are:

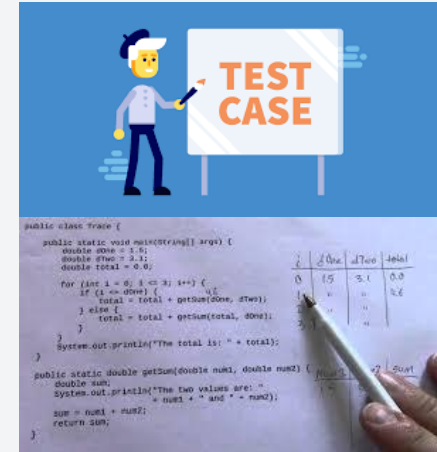
1. Faulty requirements definition
2. Client-developer communication failures
3. Deliberate deviations from software requirements
4. Logical design errors
5. Coding errors
6. Non-compliance with documentation and coding instructions
7. Shortcomings of the testing process
8. User interface and procedure errors
9. Documentation errors

Introduction

- **Terminology, IEEE 610.12-1990**
- **Fault** -- often referred to as Bug [Avizienis'00]
 - A **static** defect in software (incorrect lines of code)
- **Error**
 - An incorrect **internal** state (**unobserved**)
- **Failure**
 - **External**, incorrect behavior with respect to the expected behavior (**observed**)

Introduction

- **Testing:** Evaluating software by observing its execution
- **Debugging:** The process of finding a fault given a failure
- Testing is hard:
 - Often, **ONLY SPECIFIC INPUTS** will **trigger** the fault into creating a failure.
- Debugging is hard:
 - Given a failure, it is often difficult to know the fault.



Development of Test Cases (TCs)

Complete testing is impossible



Testing cannot guarantee the absence of faults



How to select subset of test cases from all possible test cases

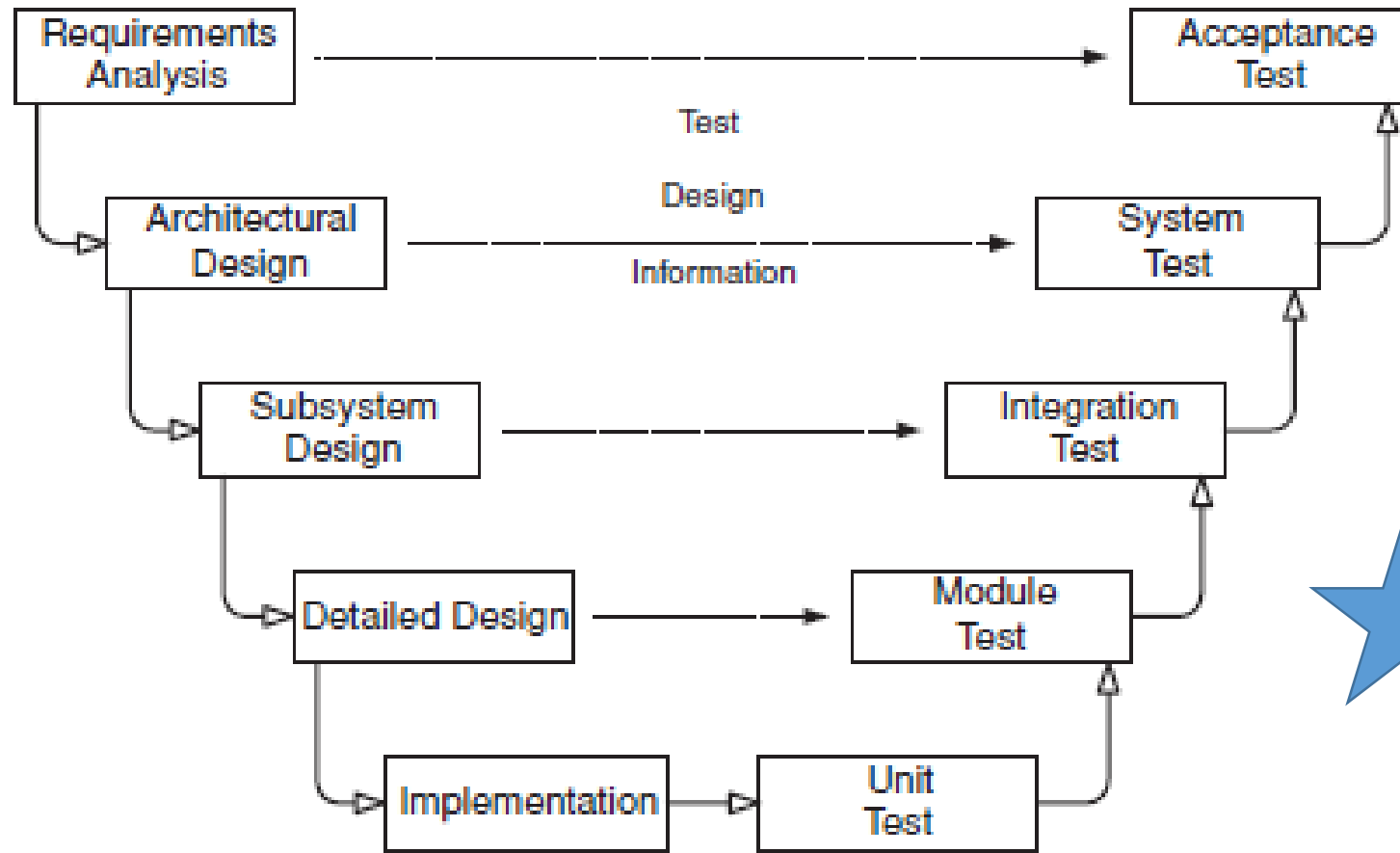
with a high chance of detecting most faults ?



Test Case Design Strategies



Testing Levels Based on Software Activity



Testing Levels Based on Software Activity

- **Acceptance Testing** – assess software with respect to requirements.
- **System Testing** – assess software with respect to architectural design.
- **Integration Testing** – assess software with respect to subsystem design.
- **Module Testing** – assess software with respect to detailed design.
- **Unit Testing** – assess software with respect to implementation.

Testing Levels Based on Software Activity

- The *detailed design* phase of **software development** determines the **structure** and **behavior** of individual modules.
- A program **unit**, or procedure, is one or more contiguous program statements, with a name that other parts of the software use to call it.
 - **Units** are called functions in C and C++, procedures or functions in Ada, **methods in Java**, and subroutines in Fortran.
- A *module* is a collection of related units that are assembled in a file, package, or class.
 - This corresponds to a file in C, a package in Ada, and **a class** in C++ and **Java**.
- *Module testing* is designed to assess individual modules in isolation, including how the component units interact with each other and their associated data structures.
- Most software development organizations make module testing the responsibility of the programmer.

Testing Levels Based on Software Activity

- Implementation is the phase of software development that actually produces code.
- *Unit testing* is designed to assess the units produced by the implementation phase and is the “lowest” level of testing.
- In some cases, such as when building general-purpose library modules, **unit testing is done without knowledge of the encapsulating software application.**
- As with module testing, most software development organizations make unit testing the responsibility of the programmer.
- It is straightforward to package unit tests together with the corresponding code through the use of tools such as *JUnit* for Java classes.

Testing Levels Based on Software Activity

- The **regression testing**, a standard part of the **maintenance phase** of software development.
- *Regression testing* is testing that is done after changes are made to the software, and its purpose is to **help ensure that the updated software still possesses the functionality it had before the updates.**

Test Case

- A test case (TC) is composed of the
 - test case values
 - expected results
 - prefix values
 - postfix values

necessary for a complete execution and evaluation of the software under test.

Test Case

- The piece of a **test case** that is referred to the most often is what we call **the test case value**.
- **Test Case Values:** The **input values** necessary to complete some execution of the software under test.
- Note that the definition of test case values is **quite broad**.
 - In a traditional batch environment, the definition is extremely clear.
 - In a Web application, a complete execution might be
 - as small as **the generation of part of a simple Web page**, or
 - it might be as complex as the completion of **a set of commercial transactions**
 - In a real-time system such as an avionics application,
 - a complete **execution might be a single frame**, or
 - it might be **an entire flight**

Test Case

- **Test case values** are the **inputs** to the program that test engineers typically focus on during testing.
- They really define what sort of testing we will achieve. However, test case values are not enough. In addition to test case values, **other inputs** are often **needed** to **run a test**.
- These inputs may depend on the source of the tests, and may be commands, user inputs, or a software method to call with values for its parameters.
- In order to evaluate the results of a test, we must know **what output a correct version of the program would produce** for that test.
- **Expected Results:** The result that will be produced when executing the test if and only if the program satisfies its intended behavior.

Test Case

- Depending on the software, the level of testing, and the source of the tests, the tester may need to **supply other inputs** to the software to affect **controllability** or **observability**.
- For example, if we are testing software for a mobile telephone, the test case values may be **long distance phone numbers**. We may also need to turn the phone on to put it in the appropriate state and then we may need to press "**talk**" and "**end**" buttons to view the results of the **test case values** and **terminate the test**.

Test Case

- **Prefix Values:** Any inputs necessary to put the software into the appropriate state to receive the test case values.
- **Postfix Values:** Any inputs that need to be sent to the software after the test case values are sent.
 - Postfix values can be subdivided into two types.
 - **Verification Values:** Values necessary to see the results of the test case values.
 - **Exit Commands:** Values needed to terminate the program or otherwise return it to a stable state.

Test Case



- **Test Case:** A test case is composed of
 - the test case values
 - expected results
 - prefix values
 - postfix values necessary for a complete execution and evaluation of the software under test.
- Coverage is a property of a set of test cases, rather than a property of a single test case.
- **Test Set:** A test set is simply a set of test cases.

Sources for TCs design

- The requirements to the program (its specification)
 - An informal description
 - A set of scenarios (use cases)
 - A set of sequence diagrams
 - A state machine
 - The program itself
 - A set of selection criteria
 - Heuristics
 - Experience

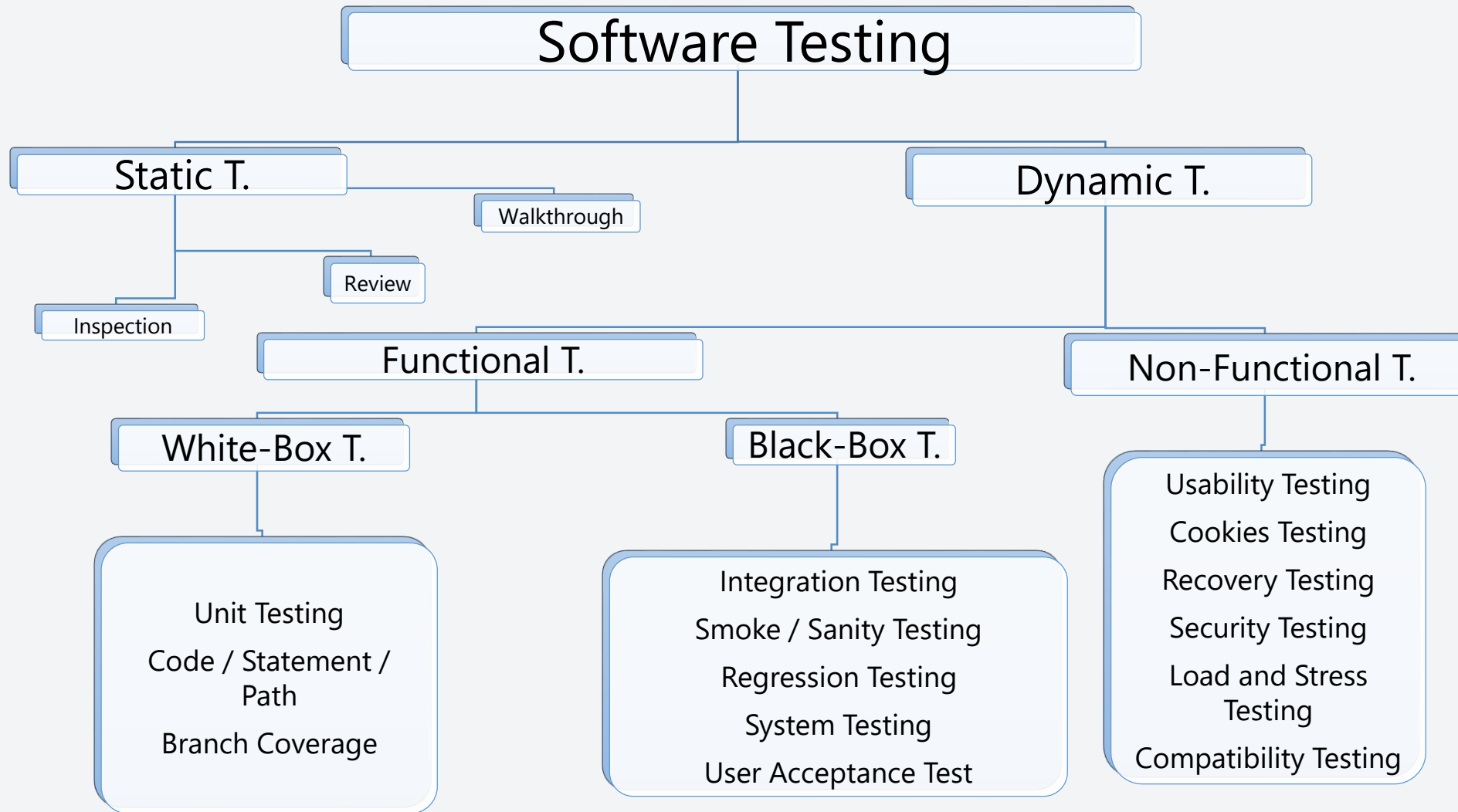




The true subject matter of the tester is not testing, but the design of test cases.

Jeff Offutt

Types of Software Testing



Types of testing

Static T.

Review

Inspection

Function

White-Box T.

Unit Testing
Code / Statement /
Path
Branch Coverage

Integration Testing
Smoke / Sanity Testing
Regression Testing
System Testing
User Acceptance Test

Cookies Testing
Recovery Testing
Security Testing
Load and Stress
Testing
Compatibility Testing

static testing

1. testing in which a test item is examined against a set of quality or other criteria without code being executed.

[ISO/IEC/IEEE 29119-1:2013 Software and systems engineering — Software testing — Part 1: Concepts and definitions, 4.42]

EXAMPLE: reviews, static analysis.

Static Testing → Walkthrough

- Go through the requirements document.
- Go through the design document.
- Perform the risk/budget analysis

Source: https://www.youtube.com/watch?v=65VslshRV_M

A blue scroll icon with a white border, containing the word "Example".

Example

Turkish Airlines - Add Membership Enrollment Details

The screenshot shows the Turkish Airlines website's 'Miles & Smiles Membership Enrollment Details' page. The browser address bar shows the URL: <https://www.turkishairlines.com/en-tr/miles-and-smiles/sign-up...>. The page header includes the Turkish Airlines logo, navigation links for 'OFFERS & DESTINATIONS' and 'FLY DIFFERENT', and a 'Sign up' button. The main heading is 'Miles & Smiles Membership Enrollment Details'. The form contains the following fields:

- First name*** *As shown in ID*: A text input field with the placeholder 'Name'.
- Surname*** *As shown in ID*: A text input field with the placeholder 'Surname'.
- Language***: A dropdown menu with the placeholder 'Select'.
- Nationality***: A dropdown menu with the placeholder 'Select'.
- Date of birth***: Three separate dropdown menus for 'day', 'month', and 'year', each with a placeholder 'day', 'month', or 'year' respectively.
- Academic title**: A dropdown menu with the placeholder 'Select'.
- Title***: Two radio buttons labeled 'Ms.' and 'Mr.', with the 'Ms.' button selected.

Example

Static Testing → Inspections

- A checklist:
 - Is the "First Name" pattern defined?
 - Is the "Surname" pattern defined?
 - What if the user is already enrolled?
 - Are the mandatory fields defined?
 - etc.

Source: https://www.youtube.com/watch?v=65VslshRV_M

dynamic testing

1. testing that requires the execution of the test item.

[ISO/IEC/IEEE 29119-1:2013 Software and systems engineering — Software testing — Part 1: Concepts and definitions, 4.9]

2. testing that requires the execution of program code

[ISO/IEC/IEEE 29119-2:2013 Software and systems engineering — Software testing — Part 2: Test processes, 4.4]

Dynamic T.

Inspection

Functional T.

White-Box T.

Unit Testing
Code / Statement /
Path
Branch Coverage

Black-Box T.

Integration Testing
Smoke / Sanity Testing
Regression Testing
System Testing
User Acceptance Test

Non-Functional T.

Usability Testing
Cookies Testing
Recovery Testing
Security Testing
Load and Stress
Testing
Compatibility Testing

Types of

white-box testing (glass-box testing)

1. testing that takes into account the internal mechanism of a system or component

Static T.

Inspection

Rev

F

onal T.

White-Box T.

Unit Testing
Code / Statement /
Path
Branch Coverage

Black-Box T.

Integration Testing
Smoke / Sanity Testing
Regression Testing
System Testing
User Acceptance Test

Usability Testing
Cookies Testing
Recovery Testing
Security Testing
Load and Stress
Testing
Compatibility Testing

black-box testing (closed-box testing)

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

software [ISO/IEC/IEEE 29119-1:2013 Software and systems engineering — Software testing — Part 1: Concepts and definitions, 4.39]

Inspection

Functional T.

White-Box T.

Unit Testing
Code / Statement /
Path
Branch Coverage

Black-Box T.

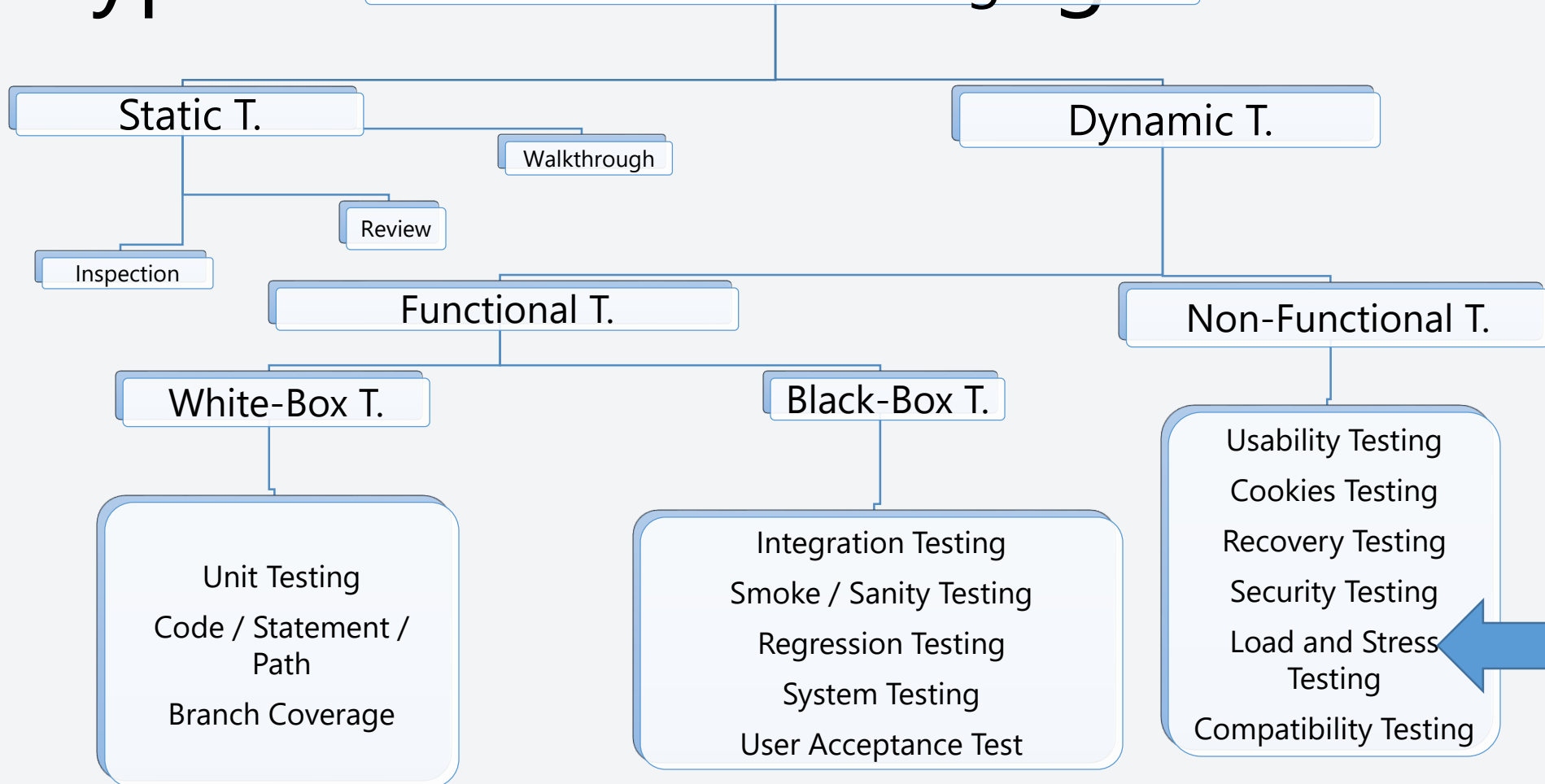
Integration Testing
Smoke / Sanity Testing
Regression Testing
System Testing
User Acceptance Test

Dynamic T.

Non-Functional T.

Usability Testing
Cookies Testing
Recovery Testing
Security Testing
Load and Stress
Testing
Compatibility Testing

Types of Software Testing



"What is HP/Loadrunner ?

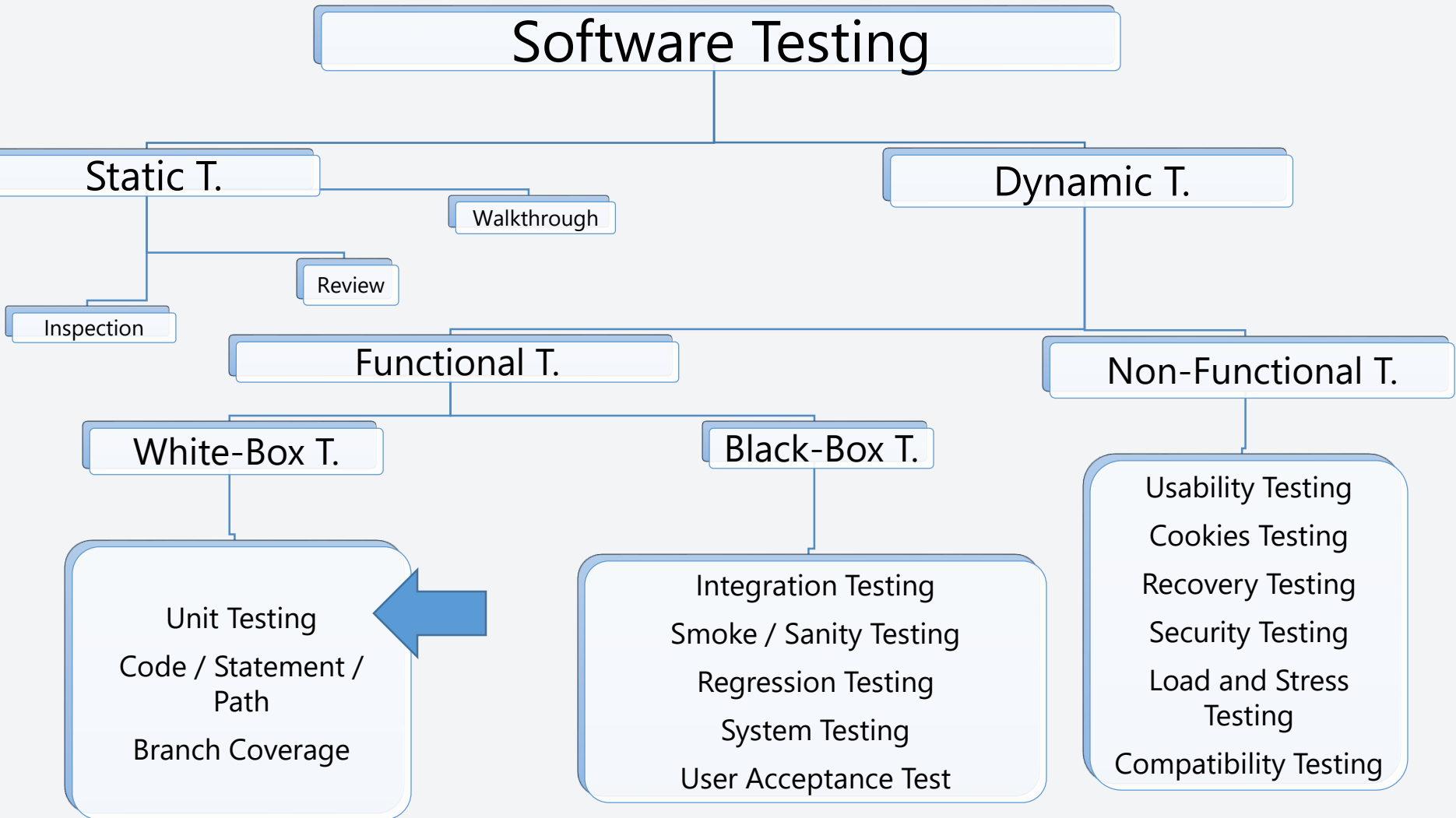
- <https://youtu.be/8L3xagVuTMQ>

Static vs. Dynamic Testing

- Some sources in the literature separate static and dynamic testing.
- **Static Testing:** Testing **without executing the program**.
 - This includes software inspections and some forms of analysis.
- **Dynamic Testing:** Testing by **executing the program with real inputs**.
- Most of the literature currently uses "**testing**" to refer to **dynamic testing** and "**static testing**" is called "**verification activities**."



Types of Software Testing



Unit Testing

- A **unit test** verifies the correctness of a specific piece of code.
 - As soon as you finish writing a piece of code, you should test it.
 - Test it as thoroughly as possible because it will get harder to fix later.

Unit Testing

- Usually unit tests apply to **methods**. You write a method and then test it. If you can, you may even want to test parts of methods.
- If you're using an OO PL, be sure to test
 - constructors
 - destructors
 - property accessors (getters or setters)
- Because unit tests are your **first chance to catch bugs**, they're extremely important.

Unit Testing

- Typically, a test is **another piece of code** that invokes the code you are trying to test and then validates the result.
 - For example, suppose you're writing a method that organizes Pokémon card decks. It groups the cards by evolution chain (cards that are related) and then **sorts the chains by their total smart ratings** (average of attack, defense, special attack, and special defense).
 - A unit test might **generate** a deck containing **100 random cards**, **pass them** into the method, and **validate** the sorted result.
- The test method could repeat the random deck test a few hundred times to make sure the sorting method works for different random decks.

Unit Testing → Regression Testing


- After you write the tests and use them to verify that your new code works, you should save the test code for later use.
- Sometimes, you may want to incorporate some or all the **unit tests** in **regression testing**.

Unit Testing

- Unit testing is commonly automated, but may still be performed manually.

Unit Testing – Junit* Example

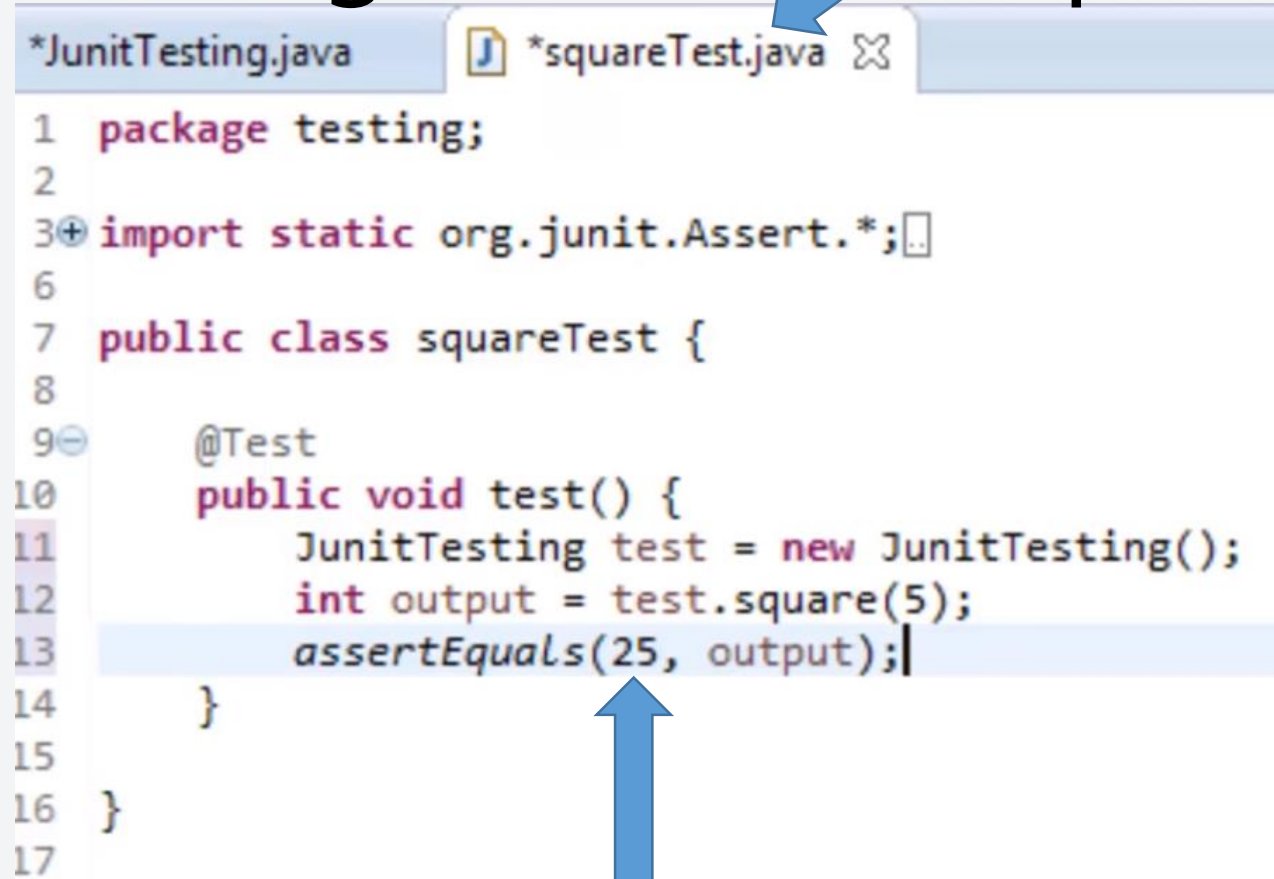
```
| *JUnitTesting.java  ✕  
1  package testing;  
2  
3  public class JunitTesting {  
4  
5  public int square(int x){  
6      return x*x;  
7  }  
8  
9  public int countA(String word){  
10     int count = 0;  
11     for(int i = 0; i < word.length(); i++){  
12         if(word.charAt(i)=='a' || word.charAt(i)=='A'){  
13             count++;  
14         }  
15     }  
16     return count;  
17 }  
18 }  
19
```



*JUnit is a unit testing framework for the Java PL

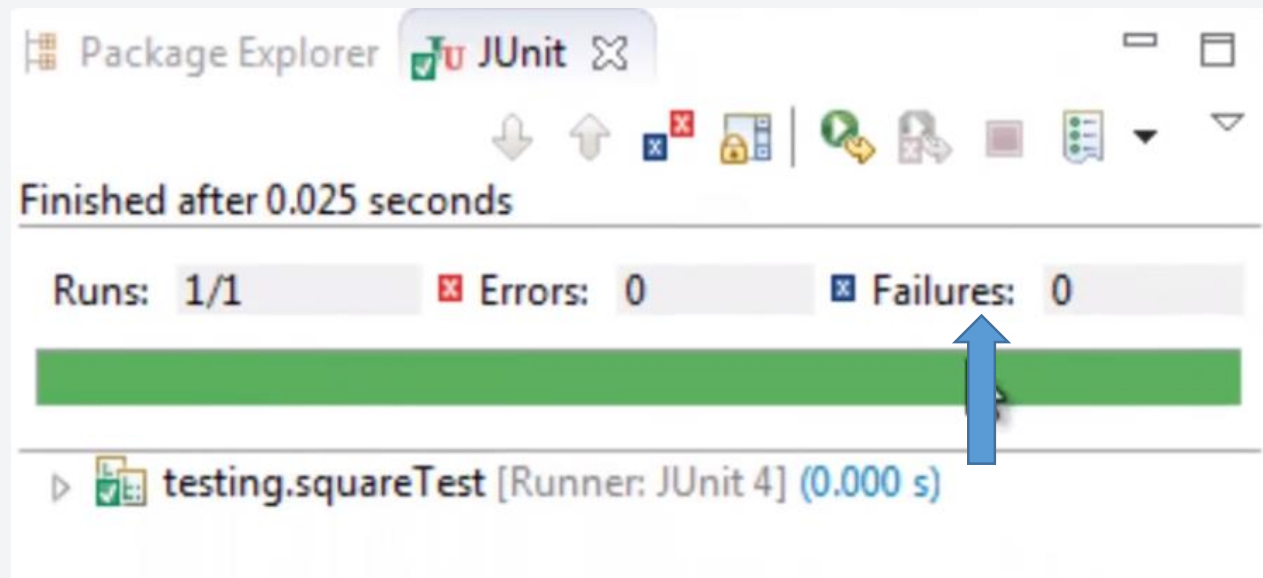
Source: <https://www.youtube.com/watch?v=l8XXfgF9GSc>

Unit Testing – Junit Example

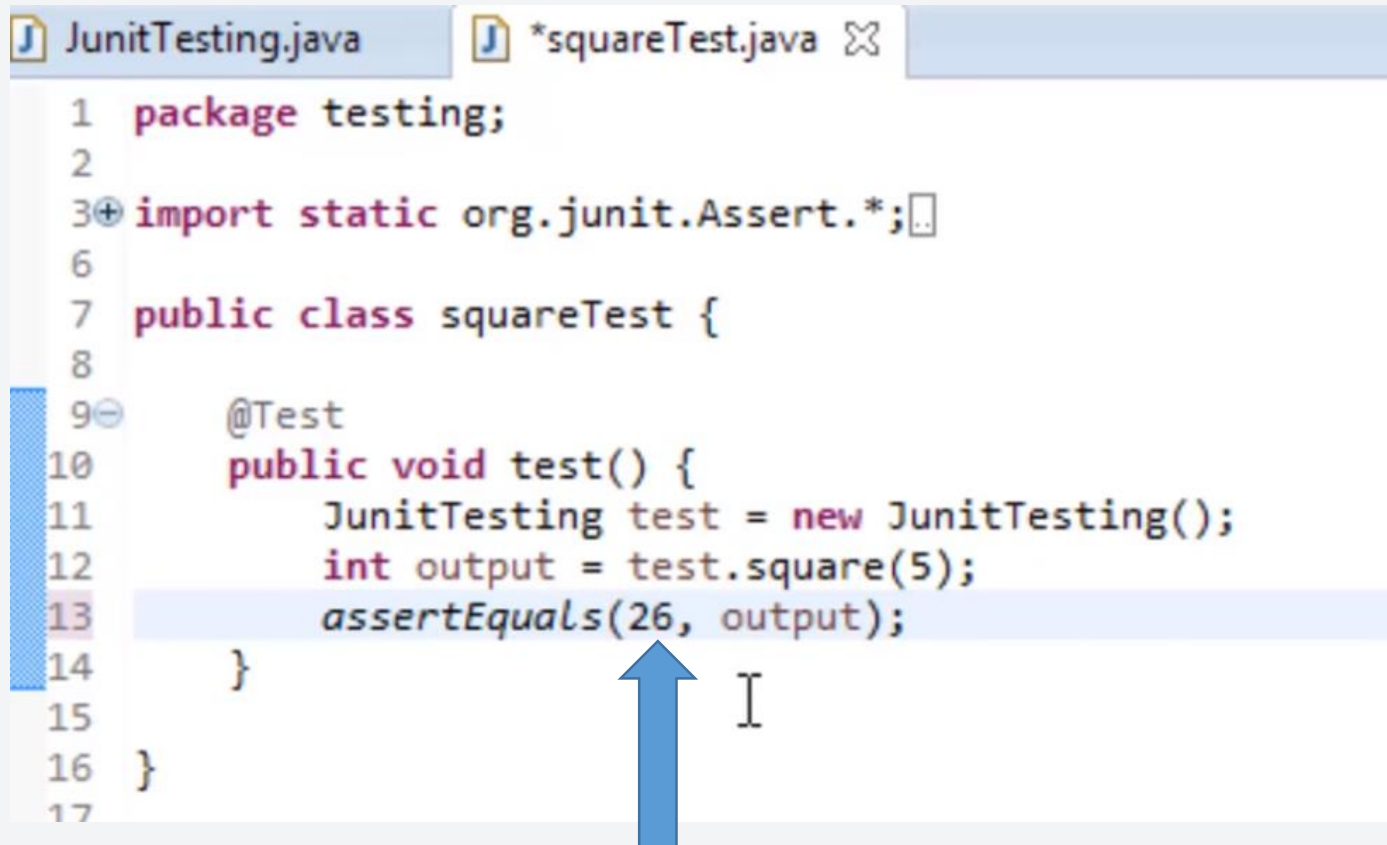


```
*JUnitTesting.java  *squareTest.java ✕
1  package testing;
2
3+ import static org.junit.Assert.*;
6
7  public class squareTest {
8
9-    @Test
10     public void test() {
11         JUnitTesting test = new JUnitTesting();
12         int output = test.square(5);
13         assertEquals(25, output);
14     }
15
16 }
17
```

Unit Testing – JUnit Example

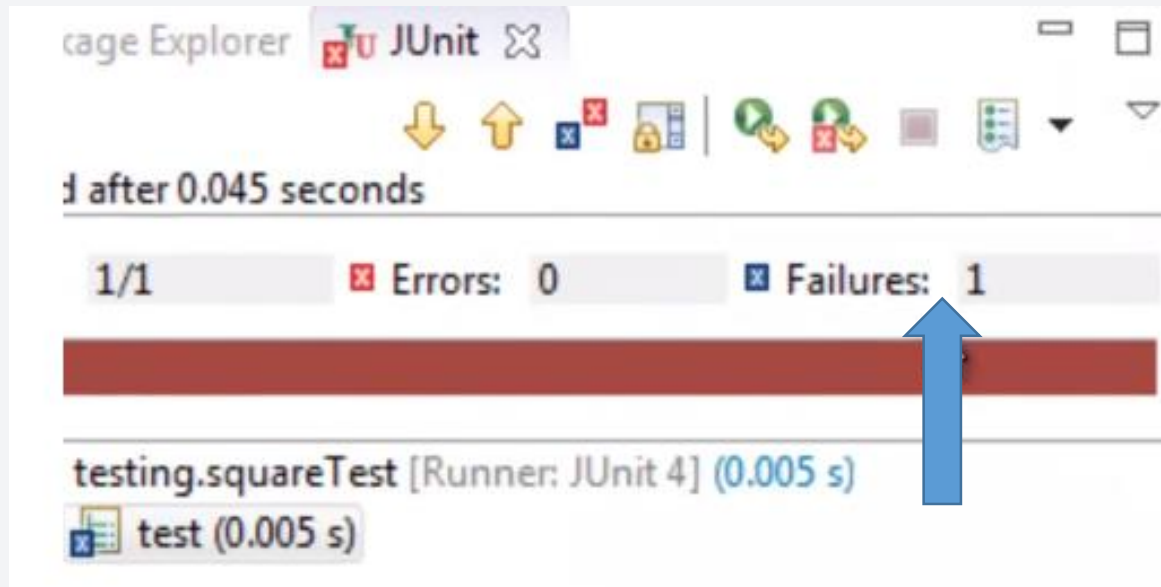


Unit Testing – Junit Example




```
1 package testing;
2
3+ import static org.junit.Assert.*;
6
7 public class squareTest {
8
9-     @Test
10     public void test() {
11         JUnitTesting test = new JUnitTesting();
12         int output = test.square(5);
13         assertEquals(26, output);
14     }
15
16 }
17
```

Unit Testing – JUnit Example

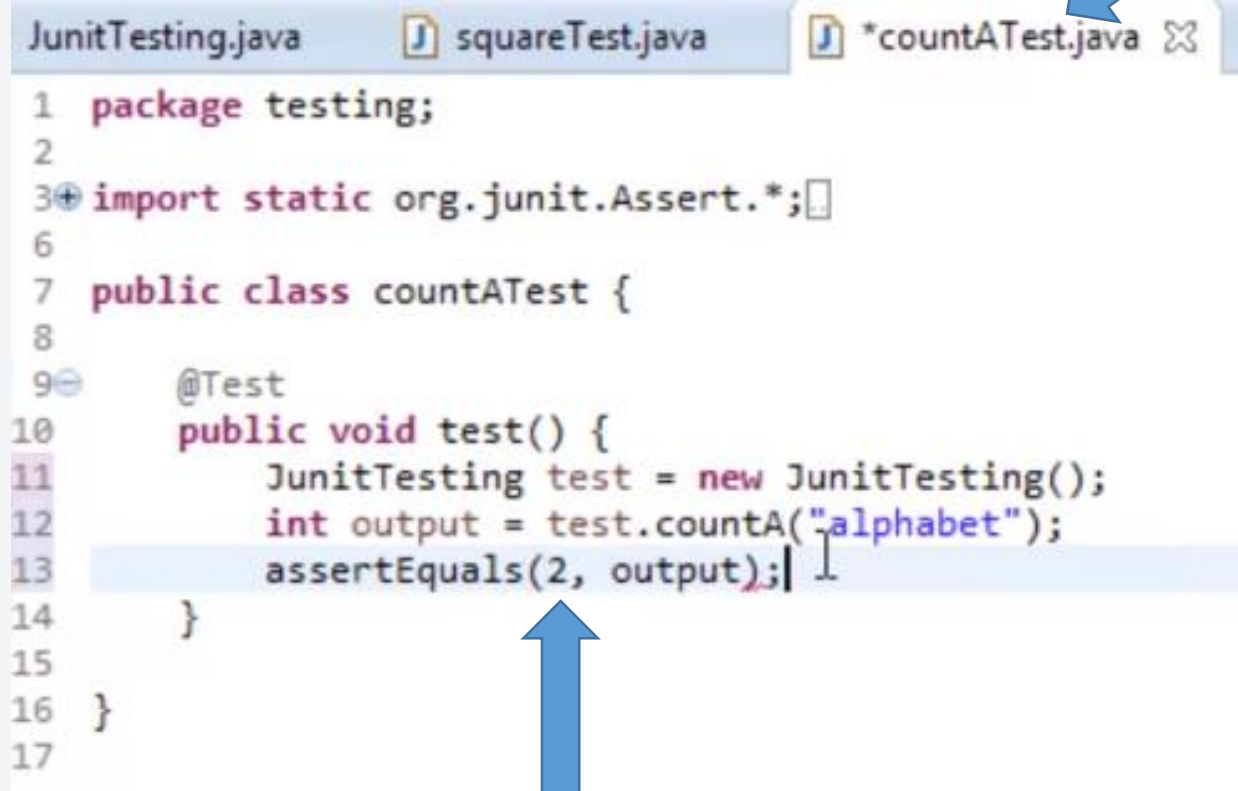


Unit Testing – Junit* Example

```
| *JUnitTesting.java ✕  
1  package testing;  
2  
3  public class JunitTesting {  
4  
5      public int square(int x){  
6          return x*x;  
7      }  
8  
9      public int countA(String word){  
10         int count = 0;  
11         for(int i = 0; i < word.length(); i++){  
12             if(word.charAt(i)=='a' || word.charAt(i)=='A'){  
13                 count++;  
14             }  
15         }  
16         return count;  
17     }  
18 }  
19
```

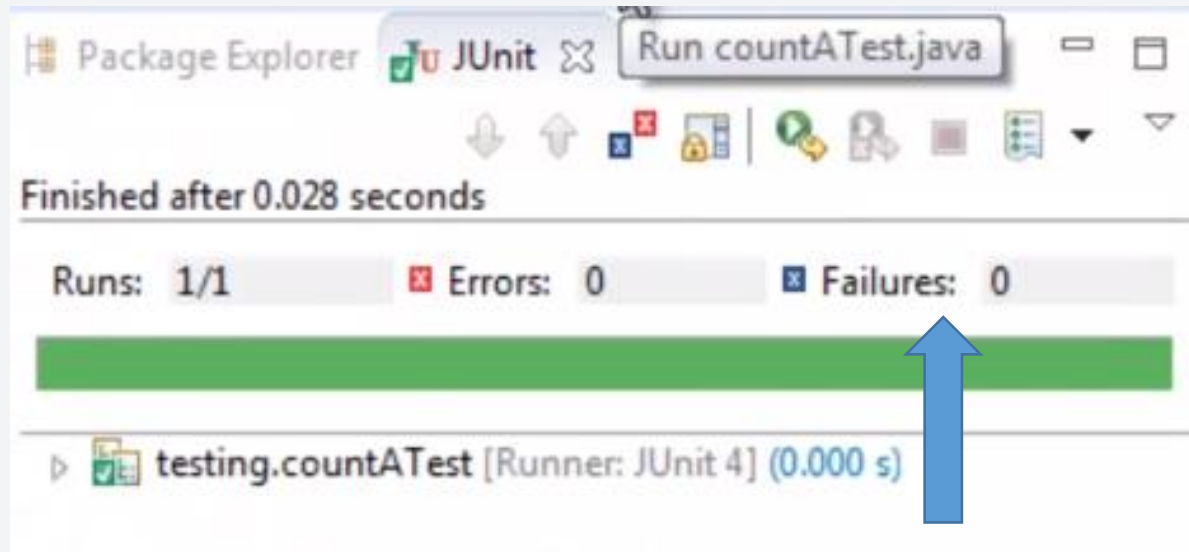


Unit Testing – Junit Example

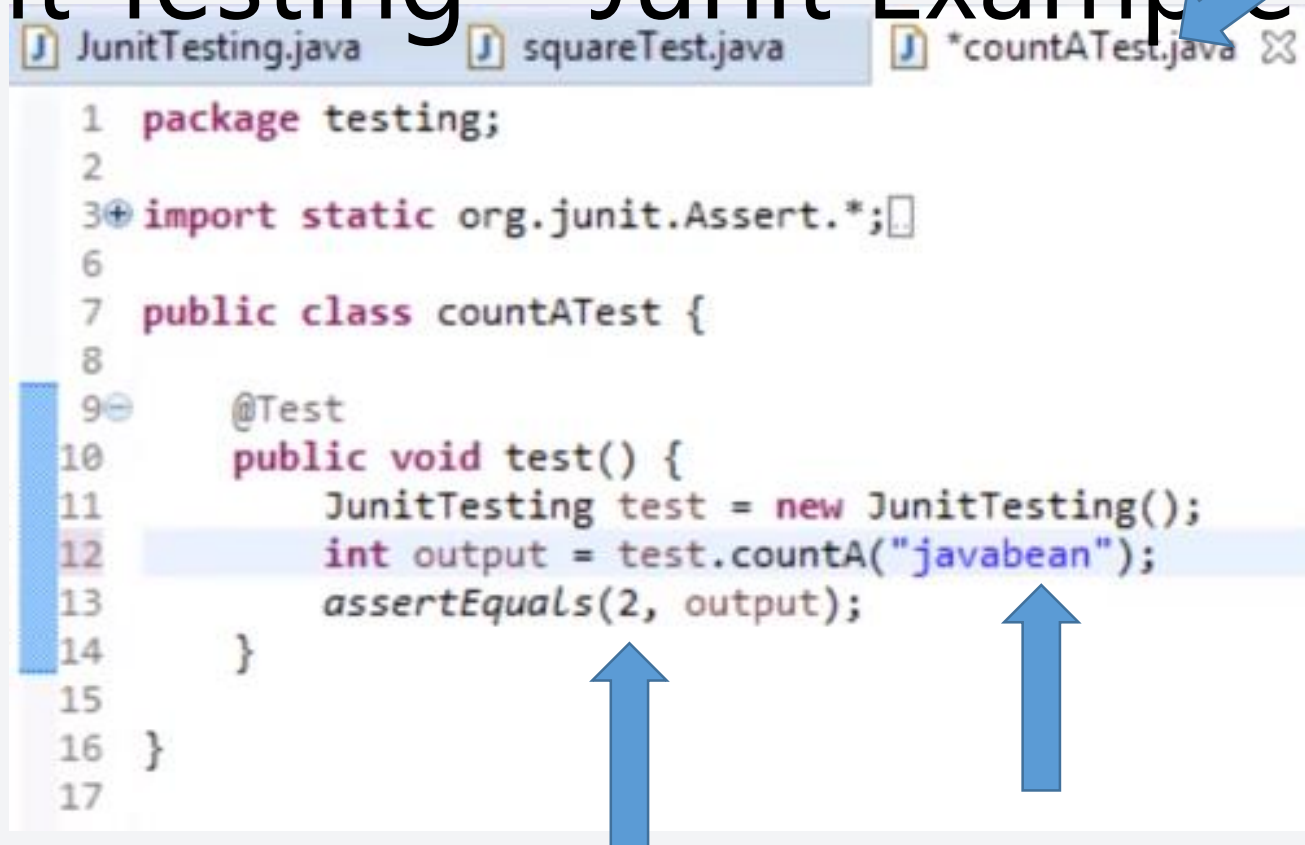


```
JUnitTesting.java  squareTest.java  *countATest.java  ⌕
1  package testing;
2
3  import static org.junit.Assert.*;
4
5
6
7  public class countATest {
8
9      @Test
10     public void test() {
11         JUnitTesting test = new JUnitTesting();
12         int output = test.countA("alphabet");
13         assertEquals(2, output);
14     }
15
16 }
17
```

Unit Testing – JUnit Example

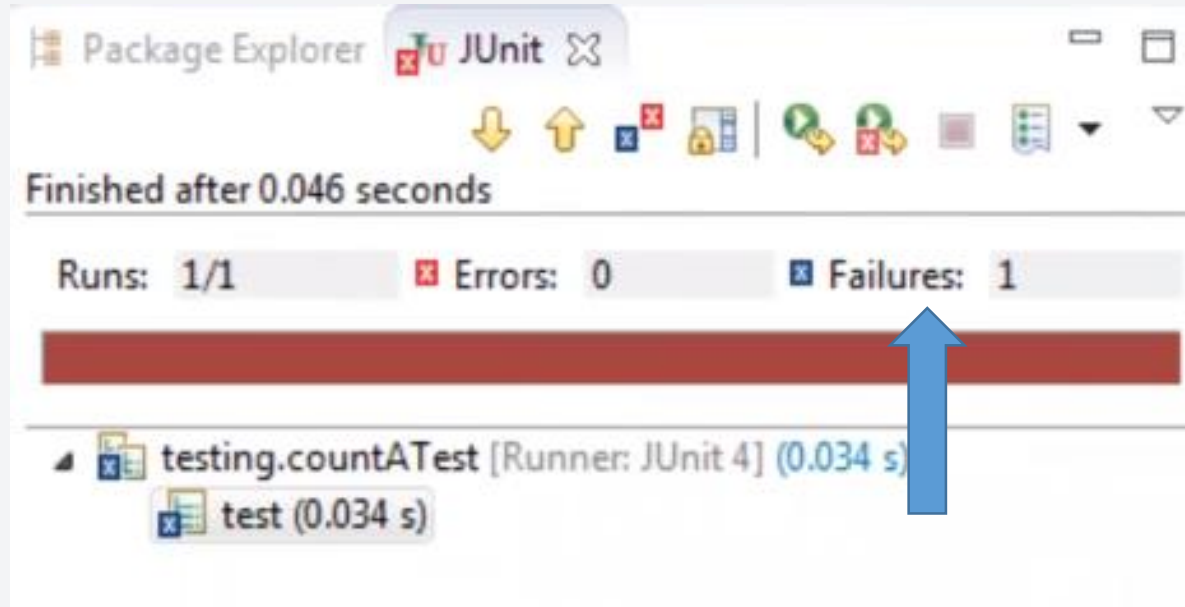


Unit Testing – Junit Example

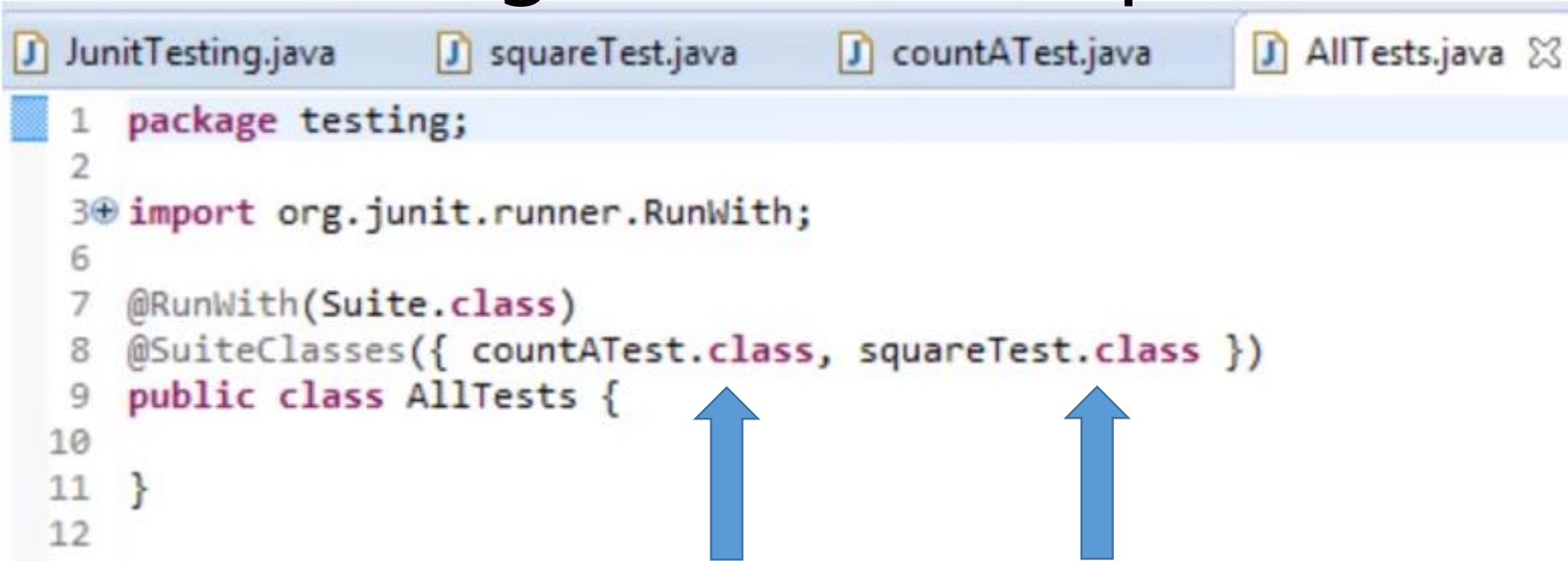


```
1 package testing;
2
3 import static org.junit.Assert.*;
4
5
6
7 public class countATest {
8
9     @Test
10     public void test() {
11         JunitTesting test = new JunitTesting();
12         int output = test.countA("javabean");
13         assertEquals(2, output);
14     }
15
16 }
17
```

Unit Testing – JUnit Example



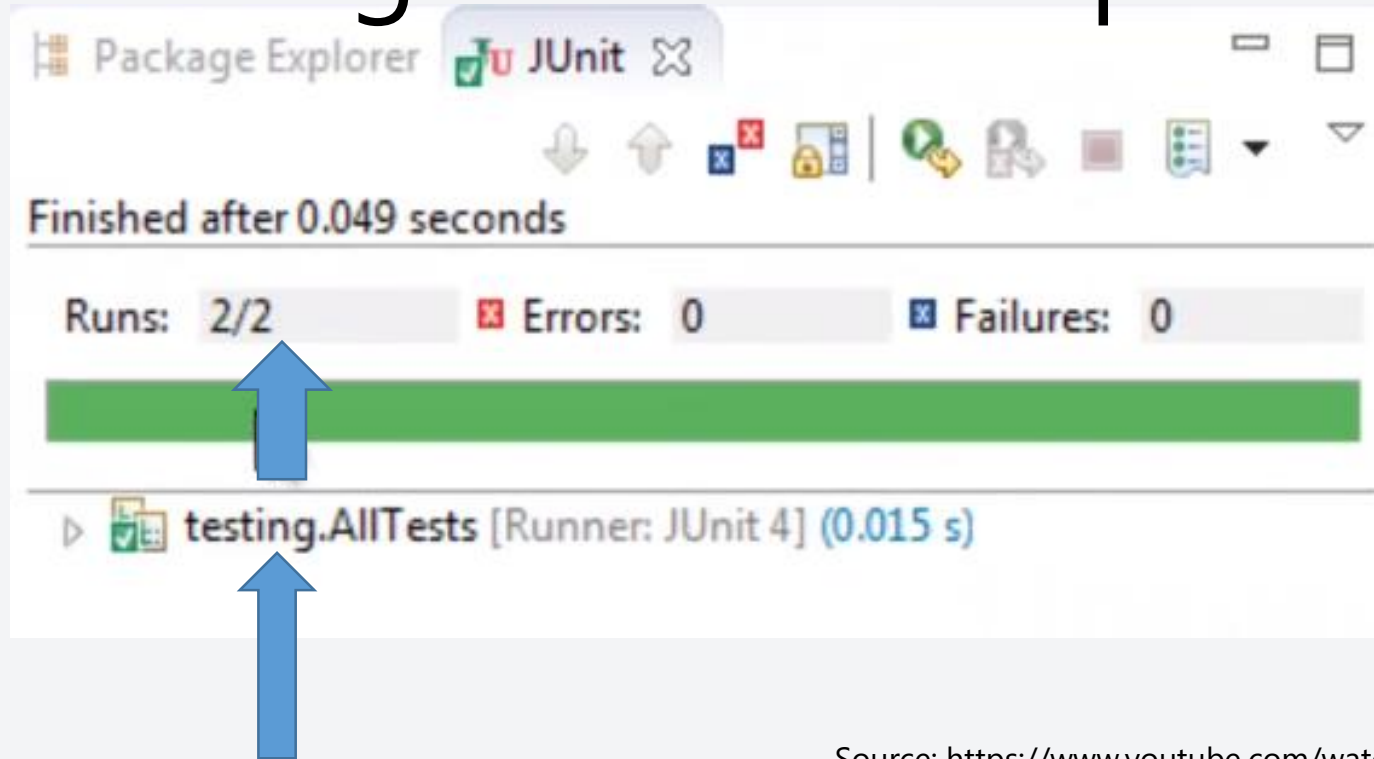
Unit Testing – Junit Example



```
1 package testing;
2
3+ import org.junit.runner.RunWith;
4
5
6
7 @RunWith(Suite.class)
8 @SuiteClasses({ countATest.class, squareTest.class })
9 public class AllTests {
10
11 }
12
```

Unit by unit ???
Again and again ???

Unit Testing – JUnit Example



Source: <https://www.youtube.com/watch?v=l8XXfgF9GSc>

Unit by unit ???
Again and again ???

Unit test

- **1.** testing of **individual routines** and **modules** by the **developer** or an **independent tester**
- **2.** test of individual programs or modules in order to ensure that there are no analysis or programming errors
 - [ISO/IEC 2382:2015, Information technology — Vocabulary]
- **3.** test of individual hardware or software units or groups of related units
 - [ISO/IEC 16350-2015 Information technology — Systems and software engineering — Application management, 4.28]

Validation vs. Verification

- **Validation:** The process of evaluating software **at the end of software development** to ensure compliance with intended usage.
 - Validation usually depends on domain knowledge; that is, knowledge of the application for which the software is written.
 - For example, validation of software for an airplane requires knowledge from aerospace engineers and pilots.
- **Verification:** The process of determining **whether the products of a given phase of the software development process fulfill the requirements established during the previous phase**.
 - Verification is usually a more technical activity that uses knowledge about the individual software artifacts, requirements, and specifications.

Verification

- 1. confirmation, through the provision of objective evidence, that specified requirements have been fulfilled
 - [ISO/IEC 12207:2008 Systems and software engineering — Software life cycle processes, 4.55; ISO/IEC 25000:2014 Systems and software Engineering — Systems and software product Quality Requirements and Evaluation (SQuaRE) — Guide to SQuaRE, 4.43; ISO/IEC TS 24748-1:2016 Systems and software engineering — Life cycle management — Part 1: Guide for life cycle management, 2.62; ISO/IEC/IEEE 15288:2015 Systems and software engineering — System life cycle processes, 4.1.54]
- 2. the evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition. It is often an internal process
 - [A Guide to the Project Management Body of Knowledge (PMBOK® Guide) — Fifth Edition]
- 3. process of evaluating a system or component to determine whether the products of a given development phase **satisfy the conditions imposed at the start of that phase**
 - [IEEE 1012-2012 IEEE Standard for System and Software Verification and Validation, 3.1.36]
- 4. process of providing objective evidence that the system, software, or hardware and its associated products conform to requirements (e.g., for correctness, completeness, consistency, and accuracy) for all life cycle activities during each life cycle process (acquisition, supply, development, operation, and maintenance), satisfy standards, practices, and conventions during life cycle processes, and successfully complete each life cycle activity and satisfy all the criteria for initiating succeeding life cycle activities
 - [IEEE 1012-2012 IEEE Standard for System and Software Verification and Validation, 3.1]

Verification

- Verification **in a life cycle context** is a set of activities that compares a product of the life cycle against the required characteristics for that product.
- This can include, but is not limited to, specified requirements, design description, and the system itself.
- **The system has been built right.**
- Verification of **interim work products** is essential for proper understanding and assessment of the life cycle phase product(s).
- A system could be verified to meet the stated requirements, yet be unsuitable for operation by the actual users.

Validation

- 1. confirmation, through the provision of **objective** evidence, that the requirements for **a specific intended use** or application have been fulfilled
 - [ISO/IEC 25000:2014 Systems and software Engineering — Systems and software product Quality Requirements and Evaluation (SQuaRE) — Guide to SQuaRE, 4.41; ISO/IEC TS 24748-1:2016 Systems and software engineering — Life cycle management — Part 1: Guide for life cycle management, 2.61; ISO/IEC/IEEE 15288:2015 Systems and software engineering — System life cycle processes, 4.1.53]
- 2. process of providing evidence that the system, software, or hardware and its associated products satisfy requirements allocated to it at the end of each life cycle activity, solve the right problem (e.g., correctly model physical laws, implement business rules, and use the proper system assumptions), and, and satisfy intended use and user needs
 - [IEEE 1012-2012 IEEE Standard for System and Software Verification and Validation, 3.1.35]
- 3. In a life cycle context, the set of activities ensuring and gaining confidence that a system is able to accomplish its intended use, goals and objectives
 - [ISO/IEC 12207:2008 Systems and software engineering — Software life cycle processes, 4.54]
- 4. process of evaluating a system or component during or **at the end of the development process** to determine whether it satisfies specified requirements
 - [IEEE 1012-2012 IEEE Standard for System and Software Verification and Validation, 3.1]
- 5. the assurance that a product, service, or system meets the needs of the customer and other identified stakeholders. It often involves acceptance and suitability with external customers
 - [A Guide to the Project Management Body of Knowledge (PMBOK® Guide) — Fifth Edition]

Validation

- Validation **in a system life cycle context** is the set of activities ensuring and gaining confidence that a system is able to accomplish its intended use, goals, and objectives.
- **The right system has been built.**
- Validation demonstrates that the system can be used by the users for their specific tasks.
- Multiple validations can be carried out if there are different intended uses.

Black-box vs. White-box testing

- **Black-box testing:** Deriving tests from external descriptions of the software, including
 - specifications
 - requirements
 - design
- **White-box testing:** Deriving tests from the source code internals of the software, specifically including
 - branches
 - individual conditions
 - statements



Black-box vs. White-box testing

- **Black-box testing:**

- (a.k.a., interface testing) is a technique in which you create test cases based only on the expected functionality of a method, class, or application without any knowledge of its internal workings.

- **White-box testing:**

- (a.k.a., : clear-box or detailed-code testing)

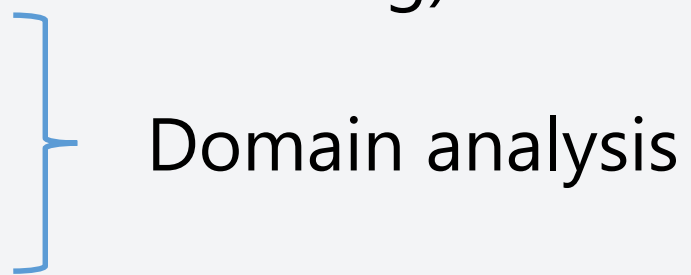
White-Box Testing

- Testing based on **program code**
 - Extent to which (source) code is executed, i.e. *Covered*
 - Different kinds of *coverage* :
 - statement coverage
 - path coverage
 - (multiple-) condition coverage
 - decision / branch coverage
 - loop coverage
 - definition-use coverage
 - etc.

White-Box testing: flow graphs

- Syntactic abstraction of source code
- Resembles classical flow charts
- Forms the basis for white box test case generation principles
- Purpose of white box test case generation: Coverage of the flow graph in accordance with one or more test criteria

Test case design techniques for Black-box testing

- Black-box testing (or functional testing):
 - Equivalence partitioning
 - Boundary value analysis
 - Cause-effect graphing
 - Behavioral testing
 - Random testing
 - Error guessing etc.
 - How to use black-box and white-box testing in combination
 - Basics: heuristics and experience
- 
- Domain analysis

Black-box: Three major approaches

- Analysis of the **input/output domain** of the program:
 - Leads to a logical partitioning of the input/output domain into 'interesting' subsets
- Analysis of the **observable black-box behavior**:
 - Leads to a flow-graph-like model, which enables application of techniques from the white-box world (on the black-box model)
- Heuristics
 - Techniques like risk analysis, random input, stress testing

Black-box: Equivalence Partitioning

- Divide ALL possible inputs into classes (partitions) such that
 - There is a finite # of input equivalence classes
 - You may reasonably **assume** that
 - the program behaves analogously for inputs in the same class
 - a test with a representative value from a class is sufficient
 - if representative detects fault then other class members will detect the same fault



Black-box: Equivalence Partitioning

- The **partition** defines the values contained in each **block** and is usually designed from knowledge of what the software is supposed to do.
- The **input domain D** is partitioned into **blocks**.
- The idea in partition coverage is that **any test** in a **block** is as good as any other for testing.
- Each partition is usually based on **some characteristic C** of the program, the program's inputs, or the program's environment.
- Some possible **characteristic** examples are:
 - Input X is null
 - Order of file F (sorted, inverse sorted, arbitrary)
 - Min separation distance of two aircraft

Black-box: Equivalence Partitioning

- Each characteristic C allows the tester to define a partition. Formally, **a partition** must satisfy two properties:
 - 1. The partition must cover the **entire domain** (completeness)
 - 2. The blocks must not overlap (disjoint)

Black-box: Equivalence Partitioning

- **Ex:** Order of file F
 - – $b1$ = Sorted in ascending order
 - – $b2$ = Sorted in descending order
 - – $b3$ = Arbitrary order
- However, this is **not** a **valid partitioning**. Specifically, if the file is of length 0 or 1, then the file will belong in all three blocks. That is, the blocks are **NOT disjoint**. ☹
- Solution: Splitting into two characteristics
- File F sorted **ascending**
 - – $b1$ = True
 - – $b2$ = False
- File F sorted **descending**
 - – $b1$ = True
 - – $b2$ = False

Black-box: Equivalence Partitioning

Strategy:

- Identify input equivalence classes
 - Based on **conditions** on **inputs** / **outputs** in **specification** / **description**
 - Both *valid* and *invalid* input equivalence classes
 - Based on heuristics and experience
 - "input x in [1..10]" → classes : $x < 1$, $1 \leq x \leq 10$, $x > 10$
 - "enumeration A, B, C" → classes : A, B, C, not{A,B,C,}
 -
- Define **one / couple** of TCs for each class
 - TCs that cover **valid** eq. classes
 - TCs that cover **at most one invalid** eq. class



Example 1: Equivalence Partitioning

- Test **a function** for calculation of **absolute value of an integer**.
- Equivalence classes :
- **Example : Equivalence Partitioning**

Condition	Valid eq. classes	Invalid eq. Classes
# of inputs	1 (A)	0, > 1 (A')
Input type	Integer (B)	non-integer (B')
particular <i>abs</i>	< 0, >= 0	

- **TCs :**
 - **TC1 (A, B) → x = -10, TC2 (A, B) → x = 100**
 - **TC3 (AB') → x = "XYZ", TC4 (AB') → x = - TC5 (A') → x = 10 20**

Example 2: Equivalence Partitioning

• Grocery Store Example

- Consider a software module that is intended to accept the **name of a grocery item** and **a list of the different sizes the item** comes in, specified in ounces. The specifications state that the **item name** is to be **alphabetic characters 2 to 15 characters** in length. Each **size may be a value in the range of 1 to 48, whole numbers only**. The sizes are to be entered in **ascending order (smaller sizes first)**. A **maximum of five sizes may be entered for each item**. The **item name is to be entered first, followed by a comma**, then followed by a list of sizes. A comma will be used to separate each size. Spaces (blanks) are to be ignored anywhere in the input.

Example 2: Equivalence Partitioning

- Derived Equivalence Classes
- 1. **Item name is alphabetic (valid)**
- 2. **Item name is not alphabetic (invalid)**
- 3. **Item name is less than 2 characters in length (invalid)**
- 4. **Item name is 2 to 15 characters in length (valid)**
- 5. **Item name is greater than 15 characters in length (invalid)**
- 6. **Size value is less than 1 (invalid)**
- 7. **Size value is in the range 1 to 48 (valid)**
- 8. **Size value is greater than 48 (invalid)**
- 9. **Size value is a whole number (valid)**
- 10. **Size value is a decimal (invalid)**
- 11. **Size value is numeric (valid)**
- 12. **Size value includes nonnumeric characters (invalid)**
- 13. **Size values entered in ascending order (valid)**
- 14. **Size values entered in nonascending order (invalid)**
- 15. No size values entered (invalid)
- 16. One to five size values entered (valid)
- 17. More than five sizes entered (invalid)
- 18. **Item name is first (valid)**
- 19. **Item name is not first (invalid)**
- 20. A single comma separates each entry in list (valid)
- 21. A comma does not separate two or more entries in the list (invalid)
- 22. The entry contains no blanks (???)
- 23. The entry contains blanks (????)

Source: <http://users.csc.calpoly.edu/~jdalbey/205/Resources/grocerystore.html>

Example 2: Equivalence Partitioning

- Black Box TCs for the **Grocery Item** Example based on the **Equivalence Classes**.

#	Test Data	Expected Outcome	Classes Covered
1	xy, 1	T	1,4,7,9,11,13,16,18,20,22
2	AbcDefghijklmno, 1, 2, 3, 4, 48	T	1,4,7,9,11,13,16,18,20,23
3	a2x, 1	F	2
4	A, 1	F	3
5	abcdefghijklmnop	F	5
6	Xy, 0	F	6
7	XY, 49	F	8
8	Xy, 2.5	F	10
9	xy, 2, 1, 3, 4, 5	F	14
10	Xy	F	15
11	XY, 1, 2, 3, 4, 5, 6	F	17
12	1, Xy, 2, 3, 4, 5	F	19
13	XY2, 3, 4, 5, 6	F	21
14	AB, 2#7	F	12

Black-box: Boundary Value Analysis

- Based on experience / heuristics :
 - Testing *boundary conditions* of **eq. classes** is more effective i.e. values directly on, above, and beneath edges of eq. classes
 - Choose input boundary values as tests in input eq. Classes instead of, or additional to arbitrary values
 - Choose also inputs that invoke *output boundary values* (values on the boundary of output classes)
 - Example strategy as extension of equivalence partitioning:
 - choose **one** (*n*) arbitrary value **in** each eq. class
 - choose values **exactly on lower** and **upper boundaries** of eq. class
 - choose values **immediately below** and **above** each boundary (if applicable)

Example: Boundary Value Analysis

- Test **a function** for calculation of **absolute value of an integer**.

Condition	Valid eq. classes	Invalid eq. Classes
particular <i>abs</i>	$x < 0, x \geq 0$	

- TCs:
 - class $x < 0$, **arbitrary value**: $x = -10$
 - class $x \geq 0$, **arbitrary value** $x = 100$
 - classes $x < 0, x \geq 0$, **on boundary** : $x = 0$
 - classes $x < 0, x \geq 0$, **below and above**: $x = -1, x = 1$

Example: Boundary Value Analysis

- TCs:

3. Item name is less than 2 characters in length (invalid)

TC1 = "a" on boundary → 1

TC2 = "" arbitrary value → 0

4. Item name is 2 to 15 (inclusive) characters in length (valid)

TC3 = "abcd" arbitrary value → 4

TC4 = "ab" on boundary → 2

TC5 = "abcdefghijklmno" on boundary → 15

5. Item name is greater than 15 characters in length (invalid)

TC6 = "abcdefghijklmnoprs" on boundary → 16

TC7 = "abcqwertyuopasdfghjk" arbitrary value → 20

More Terminology on SW Testing

- Build
- Release
- Alpha (α) Testing
- Beta (β) Testing
- (U) Acceptance Testing

release

- 1. particular version of a configuration item that is made available for a specific purpose
 - [IEEE 828-2012 IEEE Standard for Configuration Management in Systems and Software Engineering, 2.12; ISO/IEC 90003:2014 Software engineering — Guidelines for the application of ISO 9001:2008 to computer software, 3.10]
- 2. collection of new or changed configuration items that are tested and introduced into a live environment together
 - [IEEE 828-2012 IEEE Standard for Configuration Management in Systems and Software Engineering, 2.1; ISO/IEC 19770-1:2012 Information technology — Software asset management — Part 1: Processes and tiered assessment of conformance, 3.12]
- 3. collection of one or more new or changed configuration items deployed into the live environment as a result of one or more changes
 - [ISO/IEC 19770-5:2015 Information technology — IT asset management — Part 5: Overview and vocabulary, 3.28]
- 4. software version that is made formally available to a wider community
 - [IEEE 828-2012 IEEE Standard for Configuration Management in Systems and Software Engineering, 2.1]
- 5. delivered version of an application which includes all or part of an application
 - [IEEE 828-2012 IEEE Standard for Configuration Management in Systems and Software Engineering, 2.1]
- 6. set of grouped change requests, established in the Application Change Management Process, which are designed, developed, tested, and deployed as a cohesive whole
 - [ISO/IEC 16350-2015 Information technology — Systems and software engineering — Application management, 4.28]

release

- **Example:** source code, code for execution, or multiple software assets packaged into an internal production release and tested for a target platform
 - **Release management** includes **defining** acceptable quality levels for release, authority to authorize the release, and release procedures.

build

- 1. **operational version of a system** or component that incorporates **a specified subset of the capabilities** that the final product will provide
 - [IEEE 828-2012 IEEE Standard for Configuration Management in Systems and Software Engineering, 2.1; ISO/IEC 19506:2012 Information technology — Object Management Group Architecture-Driven Modernization (ADM) — Knowledge Discovery Meta-Model (KDM), 4]
- 2. **process of generating (archiving) an executable and testable system** from source versions or baselines
 - [IEEE 828-2012 IEEE Standard for Configuration Management in Systems and Software Engineering, 2.1; ISO/IEC TR 18018:2010 Information technology — Systems and software engineering — Guide for configuration management tool capabilities, 3.4]
- 3. to perform the steps required to **produce an instance of the product**
 - [IEEE 828-2012 IEEE Standard for Configuration Management in Systems and Software Engineering, 2.1]

build

- In software, this means **processing source files to derive target files**.
- In hardware, this means assembling a physical object.
- The build needs to **compile** and **link** the various versions in the correct order.
 - The build tools can be integrated into a configuration management tool.

What is the difference between build and release?

- A “**build**” is given by **development team** to the test team.
- A “**release**” is formal release of the product to **its customers**.
 - A build when tested and certified by the test team is given to the customers as “release”.
 - A “build” can be rejected by test team
 - if any of the tests fail or
 - it does not meet certain requirements
 - One release can have several builds associated with it.

Inconsistent
Terminology

Alpha (α) Testing

- **first** stage of testing before a product is considered ready for commercial or operational use
 - **often** performed **only by users within the organization** developing the software

Beta (β) Testing

- second stage of testing when a product is in limited production use
 - often performed @ a customer site

Alpha (α) and Beta (β) Testing

- Alpha testing is **also** used when developing software products that are sold as **shrink-wrapped systems**.
 - Users of these products may be willing to get involved in the alpha testing process because this **gives them early information about new system features** that they can exploit.
- Alpha testing reduces the risk that unanticipated changes to the software will have disruptive effects on their business.

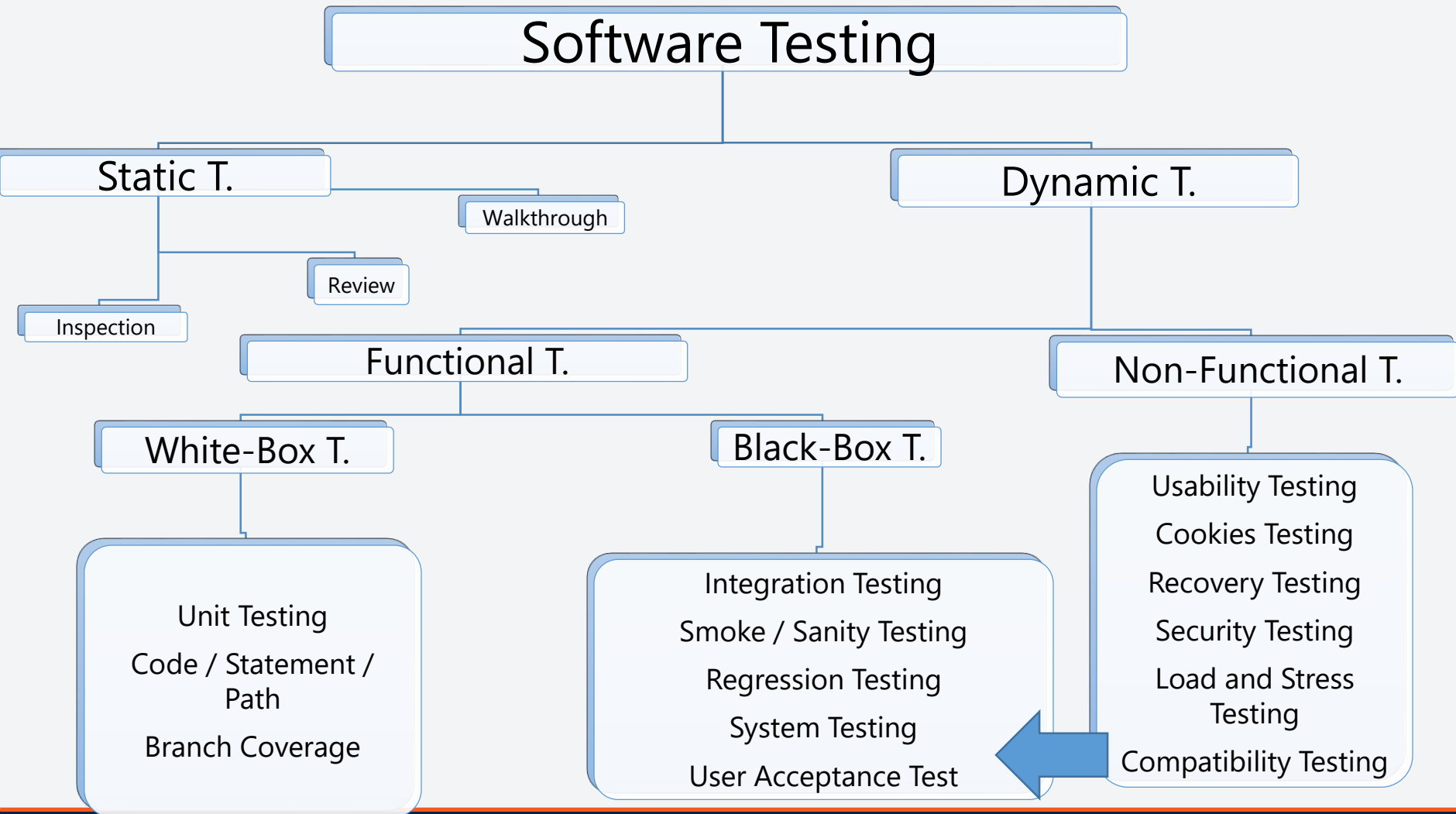
Alpha (α) and Beta (β) Testing

- When a system is to be marketed as a software product, a testing process called 'beta testing' is often used.
- **Beta testing**, where a release of the software is made available to users* to allow them to experiment and to raise problems that they discover with the system developers.
 - *a number of potential customers who agree to use that system
 - * may be a selected group of customers who are early adopters
 - Beta testing is also a form of marketing
- They report problems to the system developers. After this feedback, the system is modified and released either for further beta testing or for general sale.

Alpha (α) and Beta (β) Testing

- **Beta testing:**
- Beta testing is mostly used for software products that are used **in many different environments** (as opposed to custom systems which are generally used in a defined environment).
- It is impossible for product developers to know and replicate all the environments in which the software will be used. Beta testing is therefore essential to discover interaction problems between the software and features of the environment where it is used.

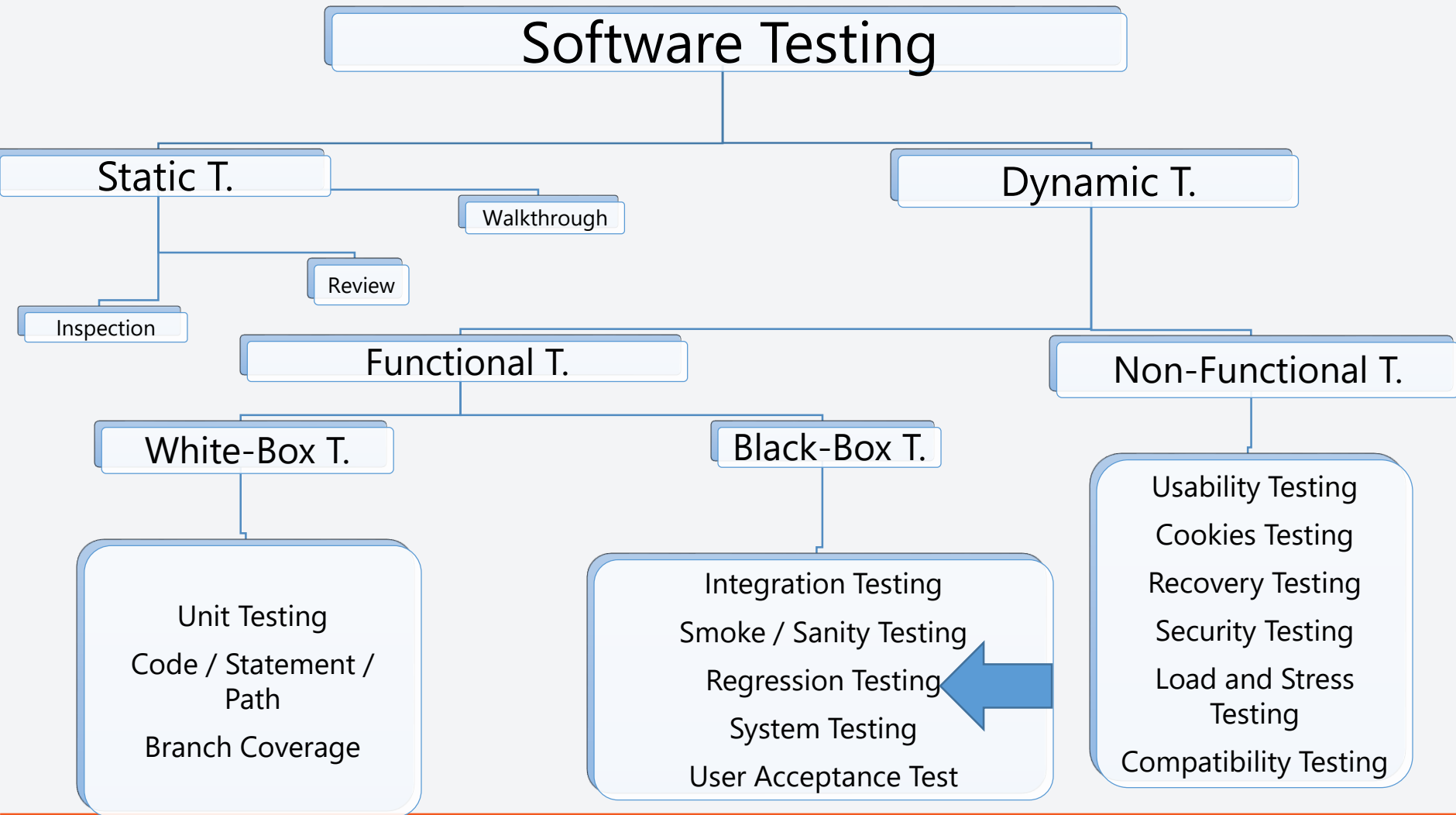
Types of Software Testing



Acceptance Testing

- Acceptance testing is an **inherent part of custom systems development**.
- It takes place after release testing.
- It involves a customer **formally testing** a system to decide whether or not it should be accepted from the system developer.
- Acceptance implies that **payment** should be made for the system.

Types of Software Testing



Regression Testing vs. Retesting

Regression Testing vs. Retesting

- **regression test**

- 1. **retesting** to detect faults introduced **by modification**

- **regression testing**

- 1. **selective retesting** of a system or component to verify that **modifications** have not caused unintended effects and that the system or component **still complies** with its specified requirements
- 2. testing required to determine that a change to a system component has not adversely affected functionality, reliability or performance and has not introduced additional defects [ISO/IEC 90003:2014 Software engineering — Guidelines for the application of ISO 9001:2008 to computer software, 3.11]
- 3. testing following modifications to a test item or to its operational environment, to identify whether regression failures occur [ISO/IEC/IEEE 29119-1:2013 Software and systems engineering — Software testing — Part 1: Concepts and definitions, 4.32]

Regression Testing vs. Retesting

- **retesting**
- confirmation testing
- 1. re-execution of test cases that previously returned a "fail" result, to evaluate the effectiveness of intervening corrective actions
[ISO/IEC/IEEE 29119-1:2013 Software and systems engineering — Software testing — Part 1: Concepts and definitions, 4.34]

Regression Testing vs. Retesting

- Most of the SWEs have confusion about the differences between regression testing and retesting.

Regression Testing

- Repeated testing of an **already tested program**, **after modification** to discover any **defects introduced** or **uncovered as a result of the changes** in the software being tested or in another **related** or **unrelated** software components.
- Usually, we do the regression testing in the following cases:
 - New functionalities are added to the application
 - Change Requirement
 - Defect fixing
 - Performance issue fix
 - Environment change
 - **Ex:** Migrating the DB from MySQL to Oracle

Retesting

- To ensure that the defects **which were found** and posted **in the earlier build** were fixed or not in the current build.
- **Ex:** Build 1.0 was released.
- Test team **found some defects**
 - Defect Id 1.0.1
 - Defect Id 1.0.2 and posted
- Build 1.1 was released, now testing the defects 1.0.1 and 1.0.2 in this build is retesting.

Regression Testing vs. Retesting

• **Example 1:**

- Bug found
- Login Page - Login button NOT working
- Login button is not working, so tester reports a bug.
- Once the bug is fixed, testers test it to make sure that whether the Login Button is working as per the expected result.

Regression Testing vs. Retesting

- **Example 1:**

- Bug found
 - Login Page - Login button NOT working
 - Login button is not working, so tester reports a bug.
 - Once the bug is fixed, testers test it to make sure that whether the Login Button is working as per the expected result.
-
- **Example 1** comes under **Retesting**.
 - Here, the tester retests the bug which was found in the earlier build by using the steps to reproduce which were mentioned in the bug report

Regression Testing vs. Retesting

- **Example 1:**

- Bug found
- Login Page - Login button NOT working
- Login button is not working, so tester reports a bug.
- Once the bug is fixed, testers test it to make sure that whether the Login Button is working as per the expected result.
- Also in the Example 1, tester tests **other functionalities** which were **related to login button** which we call as "**Regression Testing**"

Source: <https://www.youtube.com/watch?v=1p8wn29Hkr4>

Regression Testing vs. Retesting

- **Example 2:**

- Add a New Feature
 - Login Page - Added "Stay signed in" checkbox
 - Tester tests the new feature to ensure whether the new feature is working as intended.
-
- **Example 2** comes under **Regression Testing**.
 - Here, the tester tests the new feature (Stay signed in) + also tests **the relevant functionalities**.
 - Testing the relevant functionalities while testing the new feature come under Regression Testing.

Regression Testing vs. Retesting

- **Example 3:**

- Imagine, an app under test has 3 modules namely, Admin, **Purchase**, and **Finance**.
 - **Finance** module **depends** on **Purchase** module.
- If a tester found a bug on a **Purchase** module and posted. Once the bug is fixed, the tester needs to do **retesting** to verify whether the bug related to the **Purchase** is fixed or not **AND** also the tester needs the **regression testing** to test the **Finance** module which depends on the **Purchase** module.