Chapter 7: **Testing**

**- Reliability:** how closely the system conforms to expected behavior

**- Software Reliability:** probability that a software system will not cause failure for the specified time under specified conditions

It is measured by Mean time to failure (MTTF), the length of time a device or other product is expected to last in operation

**- Error**: incorrect software behavior ex: message box text said "Welcome null."

**- Fault**: algorithmic cause of error ex: account name field is not set properly. A fault is not an error, but it can lead to it

**Quality Control techniques**:

**Fault Avoidance:** prevent errors by finding faults before the system is released

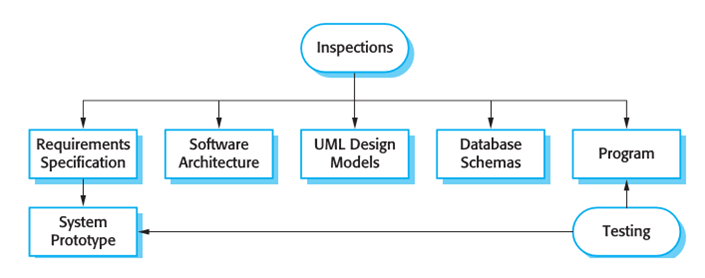
* **Review:** manual inspection **Walkthrough:** developer presents code to team **Inspection:** team looks at code without guidance

**Fault Detection**: find existing faults without recovering from the errors

* **Debugging**: move through steps to reach error **Testing**: tries to expose errors in planned way

Ideally, every possible input to a system should be tested which is expensively time-consuming

**Fault Tolerance**: when system can recover from failure by itself



**- Verification: "Are we building the product right**” The software should conform to its specification

**- Validation: "Are we building the right product”** The software should do what the user really requires

**- Inspection:** analysis of the static system representation to discover problems

Do not require execution and may be applied to any representation of the system

Can find errors in incomplete systems and is static thus it can find masked errors

Cannot check non-functional characteristics so it’s coupled with testing during V&V

**- Testing:** exercising and observing product behaviour

* **Types: - Development testing:** system is tested during development to discover bugs and defects
* **Unit Testing**: looks for errors in objects or subsystems
* **Integration Testing**: find errors with connecting subsystems together
* **System Structure Testing**: integration testing all parts of system together
* **System Testing**: test entire system behavior as a whole, with respect to scenarios and requirements
* **Functional Testing**: test whether system meets requirements
* **Performance Testing**: non-functional requirements, design goals
* **Acceptance / Installation Testing**: done by client

**- Release testing:** separate testing teams test a complete version of the system before it is released to users

**- User testing:** users or potential users of a system test the system in their own environment

* **Black-box Test:** focuses on input/output of each component

**White-box Test:** focuses on internal states of objects

* **Regression Testing**: re-executing all prior tests after a code change

Often automated by scripts to ensure that fixed bugs are still fixed as new feature/fix can cause a new bug or reintroduce an old one

* **Unit testing:** looking for errors in individual objects or subsystems in isolation

1. Reduces number of things to test 2. Easier to find faults when errors occur 3. Can test all components in parallel

* **Types of Integration Testing:**
  + **Big Bang**: no stubs; do unit testing, then throw all parts together + faster - costly/ error-prone

**Bottom-Up**: integrate upward into double, triple, quadruple test + fewer stubs - tests UI last

* + **Top-Down**: test top layer (UI) first, then add layers to replace underlying stubs + focuses on UI - needs many stubs
  + **Sandwich integration testing:**

Perform top-down and bottom-up testing at **same time**

**Modified sandwich: test each layer individually then do the sandwich**

* + - bread (UI)
    - meat (major subsystems)
    - bread (ground layer)

JUnit

* **Junit 3- (Old Style):**

**public** ***YourTestingClassName*** *extends* **TestCase***{*

**//Class must inherit a specific super class called TestCase**

* **public** ***YourClassName***() // constructor
  + **//run once at start of ALL tests (when testing object is created)**
* **public** void **setUp**()
  + **//run at start of each test**
* **public** void **tearDown**()
  + **//run at end of each test**

**public** void **testXYZ**()

* + **//All test methods should start with “test”**

}

* **Junit 4+-:**

**import org.junit.\*;**

**import static org.junit.Assert.\*;**

public class ***YourTestingClassName***{

**@Test**

public void ***YourTestName***() {

**//Methods with a @Test annotation are flagged as JUnit test cases, run when JUnit runs your class**

}

**@Before**

public void ***SomeFunction***() {

**//Run before all tests**

}

**@After**

public void ***AnotherFunction***() {

**//Run after all tests**

}

}

**//Use these inside the test case:**

* **public** void **assertTrue**(boolean condition)  
  **public** void **assertTrue**(*String message*, boolean condition)
* **public** void **assertFalse**(*String message*, boolean condition)
* **public** void **assertEquals**(*String message*, Object expected, Object actual)  
  **public** void **assertNotEquals**(*String message*, Object expected, Object actual)
* **public** void **assertSame**(*String message*, Object expected, Object actual)  
  **public** void **assertNotSame**(*String message*, Object expected, Object actual)  
   **//these methods compare using ==, not .equals**
* **public** void **assertNull**(*String message*, Object obj)  
  **public** void **assertNotNull**(*String message*, Object obj)
* **//forcibly causes the test to fail**
* **public** void fail(*String message*)
* **Junit 5+-:**

**import org.junit.\*;**

**import static org.junit.Assert.\*;**

public class ***YourTestingClassName***{

**@Test**

void ***YourTestName***() {

**//Methods with a @Test annotation are flagged as JUnit test cases, run when JUnit runs your class**

}

**//After @Test we can add:**

**@DisabledOnOs(WINDOWS)**

**@EnabledOnOs({ LINUX, MAC })**

**@EnabledOnJre(JAVA\_8)**

**@BeforeEach**

void ***SomeFunction***() { **//Run before each test** }

**@AfterEach**

void ***AnotherFunction***() { **//Run after each test** }

**@BeforeAll**

void ***SomeFunction2***() { **//Run before all tests** }

**@AfterAll**

void ***AnotherFunction2***() { **//Run after all test**s }

}

**//Same as Junit4 with more flexibility:**

* **public void assertTrue(boolean *condition*)**
* **public void assertTrue(*String message*, boolean *condition*)  
  public void assertTrue(boolean *condition, String message*)  
  public void assertTrue(boolean *condition, Supplier<String> messageSupplier*)**
* **public void assertTrue(BooleanSupplier *conditionSupplier*)**
* **public void assertTrue(BooleanSupplier *conditionSupplier, String message*)**

**public void assertTrue(BooleanSupplier *conditionSupplier, Supplier<String> messageSupplier*)**

**//Lazy assetion evaluations examle:**

* **@Test**
* **void dependentAssertions() {**
* **// within a code block, if an assertion fails the subsequent code in the same block will be skipped.**
* **assertAll("properties",**
* **() -> {**
* **String firstName = person.getFirstName();**
* **assertNotNull(firstName);**
* **// Executed only if the previous assertion is valid.**
* **assertAll("first name",**
* **() -> assertTrue(firstName.startsWith("J")),**
* **() -> assertTrue(firstName.endsWith("n"))**
* **);**
* **},**
* **() -> {**
* **// grouped assertion, so processed independently of results of first name assertions.**
* **);**
* **}**
* **);**
* **}**

**Performance Test Example (Junit 3-):**

**long expectedTimeout = 5000; // 5s**

**long start = System.currentTimeInMillis();**

**callMethodToTestPerformance(…);**

**long end = System.currentTimeInMillis();**

**assertTrue(end-start < expectedTimeout);**

* **Performance Test Example (Junit 4+):**

**@Test(timeout = 5000)**

**public void *YourTestName*() {**

**....**

**}**

* **Performance Test Example (Junit 5+):**

**@Test**

**public void *YourTestName*() {**

**assertTimeout(ofSeconds(5),**

**() -> {**

**// ... Code that may take so long**

**} , () -> "The above code took more than 5secs");**

**assertTimeoutPreemptively(ofSeconds(5),**

**() -> {**

**// ... Code that may take so long**

**} , () -> "The above code took more than 5secs & was aborted");**

**}**

* **Exception Test Example (Junit 3-):**

**try{**

**callMethodThatShouldMakeAnException();**

**fail("If we got here then the exception wasn’t thrown");**

**}**

**catch(SomeException e){**

**assertTrue("Exception was thrown normally", 1==1);**

**}**

* **Exception Test Example (Junit 4+):**

**@Test(expected = SomeException.class)**

**public void *YourTestName*() {**

**callMethodThatShouldMakeAnException();**

**}**

* **Exception Test Example (Junit 5+):**

**@Test**

**public void *YourTestName*() {**

**assertThrows(SomeException.class,**

**()-> {**

**callMethodThatShouldMakeAnException();**

**});**

**}**

* **Test Suites (Junit 5+): A separate class that runs a set of test classes together**

**import org.junit.runner.\*;**

**import org.junit.runners.\*;**

**@RunWith(Suite.class)**

**@Suite.SuiteClasses({**

***YourTestingClassName*.class,**

***YourOtherTestingClassName*.class**

**})**

**public class *YourSuiteClassName* {}**