# CSE 464 Software Quality Assurance and Testing

## *Software Testing - JUnit*

**Xusheng Xiao**

Associate Professor
School of Computing and Augmented Intelligence
Arizona State University

# JUnit

- Java unit testing framework
- Already bundled with Eclipse
- Defines format for writing test cases (executable Java), manages execution of test cases, result comparison, and easy-to-read evaluation report

# JUnit Basics

- **Test case**: a single method (**annotated with `@Test`**) that JUnit can execute
  - Should have only 1 set of input values
  - Pass/fail usually determined by an assert statement
- **Test class**: a Java class that contains JUnit test cases
  - Use for focused testing on a single Java class or set of classes
  - Name it like a normal Java class but add "Test" to the name
- **Test suite**: one or more test classes

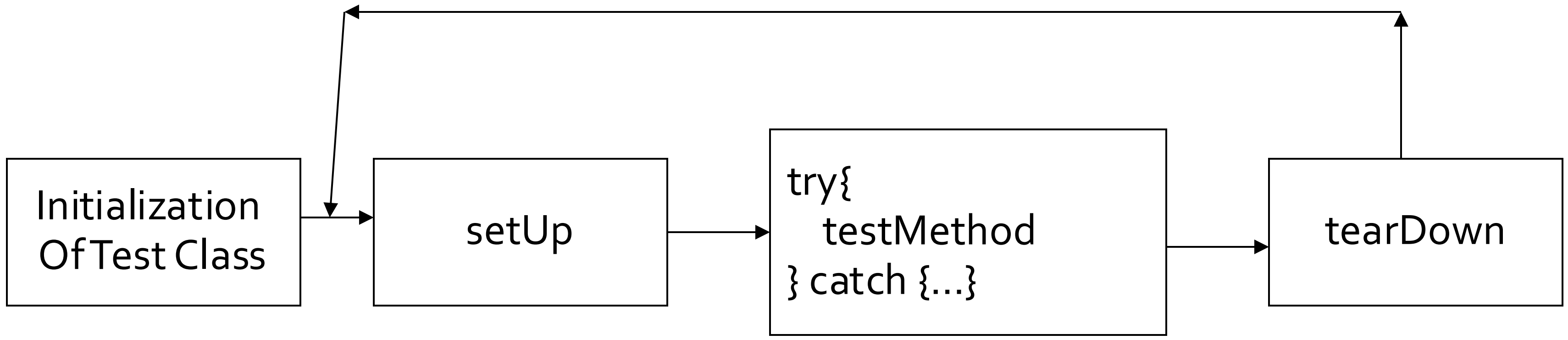
## JUnit Basics

- assert family of methods do the testing between expected and actual values
- A test passes or fails based on outcome of assert statement
- **But**: a test can also pass without an assert statement
    - E.g., test passes if an exception is thrown

## Note

- **Test case execution order is arbitrary**; thus no test can depend on the outcome of another test
- A helpful resource is http://www.vogella.com/tutorials/JUnit/article.html
  - Some of this material and organization was borrowed from Mr. Vogella

# State machine of JUnit



## asserts

- General format: `assert...([message,] <expected>, <actual>);`
- **`assertTrue`**`,` **`assertFalse`**`([msg,]<boolean result>)`
- **`assertEquals`**`([msg,]<exp>,<act>[,tolerance])`
- **`assertNull`**`,` **`assertNotNull`**
- **`assertSame`**`,` **`assertNotSame`**`: compares references`

## Assertions

```
public void testCapacity() {        // a test method
    ….
    assertTrue(fFull.size() == 100+size);  //assertion
}
```

If assertion fails:

**Assertion failed: myTest.java:150 (expected true but was false)**

Not so good!

Try: assertEquals(100+size, fFull.size()); //expected value first

**Assertion failed: myTest.java:150 (expected 102 but was 103)**

Better!

Try: assertEquals(“list length”, 100+size, fFull.size());

**Assertion failed: myTest.java:150 (list length expected 102 but was 103)**

## Fail

- `fail([String msg])`
- A method that causes the test to fail with the provided message
- Default statement added to new test cases that Eclipse creates
- Normal starting point for tests when unit has not yet been created (e.g., Test-Driven Development)

## Comparing Objects

- **assertEquals**: custom objects need to implement equals; otherwise compares references
- **assertArrayEquals**: compares contents of arrays

# Exception Testing

- Exceptions are **NOT** errors
- Exceptions are mechanisms to **handle errors** and **avoid failures**
- Good programming defines/handles its own exceptions
  - Otherwise the JVM handles them and will make you look bad
- Thus, <u>**very important**</u> to also test your exception handling

## JUnit Exception Testing

- Ways to do this:
  - try/catch with assert
  - @Test (expected=Exception)

## try/catch with assert

- Put a try/catch in the test method
- Put code to trigger exception in try code block and fail() at end of the code block
  - Thus, if exception is NOT thrown, test fails
- Put assert in catch code block to test property of exception when it is thrown.

## @Test (expected=

- Add `(expected=<exception name>.class)` to `@Test annotation`
- Test will **<u>only</u>** pass if an exception of that type is thrown
- Any exception of that type will pass
  – Cannot check the exception’s message

## Timeout Tests

- Use `@Test (timeout = <ms>)` annotation
- Test will fail if asserts in test fail OR timeout occurs
- Great for system/performance tests
  - E.g., test if system can perform 10 transactions per second

## Setup Methods

- **`@Before`** methods: execute before EACH test; useful for preparing/resetting test data
- **`@BeforeClass`** methods: execute once before ALL tests; useful for initializing time-consuming tasks like opening files, database connections, etc. (methods must be static)

## Tear down Methods

- **`@After`** methods: execute after EACH test; useful for deleting temporary data, releasing resources
- **`@AfterClass`** methods: execute once after ALL tests; useful for closing files, database connections, etc. (methods must be static)

# Tear down

Consider the following test code

```
void setUp() {
    File f = open(“foo”);
    File b = open(“bar”);
}
void testAAA() {
    use f and b
}
void testBBB(){
    use f and b
}
```

Problems?

Better?

```
void setUp() {
    File f = open(“foo”);
    File b = open(“bar”);
}
void testAAA() {
    try {
        use f and b
    } finally {
        clean&close f, b
    }
}
void testBBB() {
    try {
        use f and b
    } finally {
        clean&close f, b
    }
}
```

# Tear down

Consider the following test code

```
void setUp() {
    File f = open(“foo”);
    File b = open(“bar”);
}
void testAAA() {
    use f and b
}
void testBBB(){
    use f and b
}
void tearDown{
    clean&close f, b
}
```

```
void setUp() {
    File f = open(“foo”);
    File b = open(“bar”);
}
…
void tearDown{
   try{
        clean&close f
    }catch{
        …
    }
    the same for b
}
```

Problems?

## Tear down

- Be careful about tear down
  - If tear down is not complete, a test failure may affect the following test cases
  - Recover the changes done to global data that are not well handled by the setup
    - Database, files, network, global variables
  - Clean resources
  - Caution of exceptions in tear down itself

## What To Test?

- **Test complexity well**
  - More complexity -> more defects

- What Not to Test?
  - Getters and Setters (unless they are complex)

## Testing: Concepts

- Test case
- Test oracle
- Test suite
- Test script
- Test driver
- Test coverage

## Testing: Concepts

- Test case
  - An execution of the software with a given list of input values
  - Include:
    - Input values, sometimes fed in different steps
    - Expected outputs
- Test oracle
  - The expected outputs of software by feeding in a list of input values
  - A part of test cases
  - Hardest problem in auto-testing: test oracle problem

# Testing: Concepts: Example

| | |
|---|---|
| *ID* | 14 |
| *Title* | Add customer |
| *Pre-Conditions* | Sign in with sales authorization |
| *Test Steps* | 1. Select the client module.<br>2. Enter the customer information.<br>3. Click "Add". |
| *Expected Results* | A message appears in the program's status bar. The message reads "New customer added". |

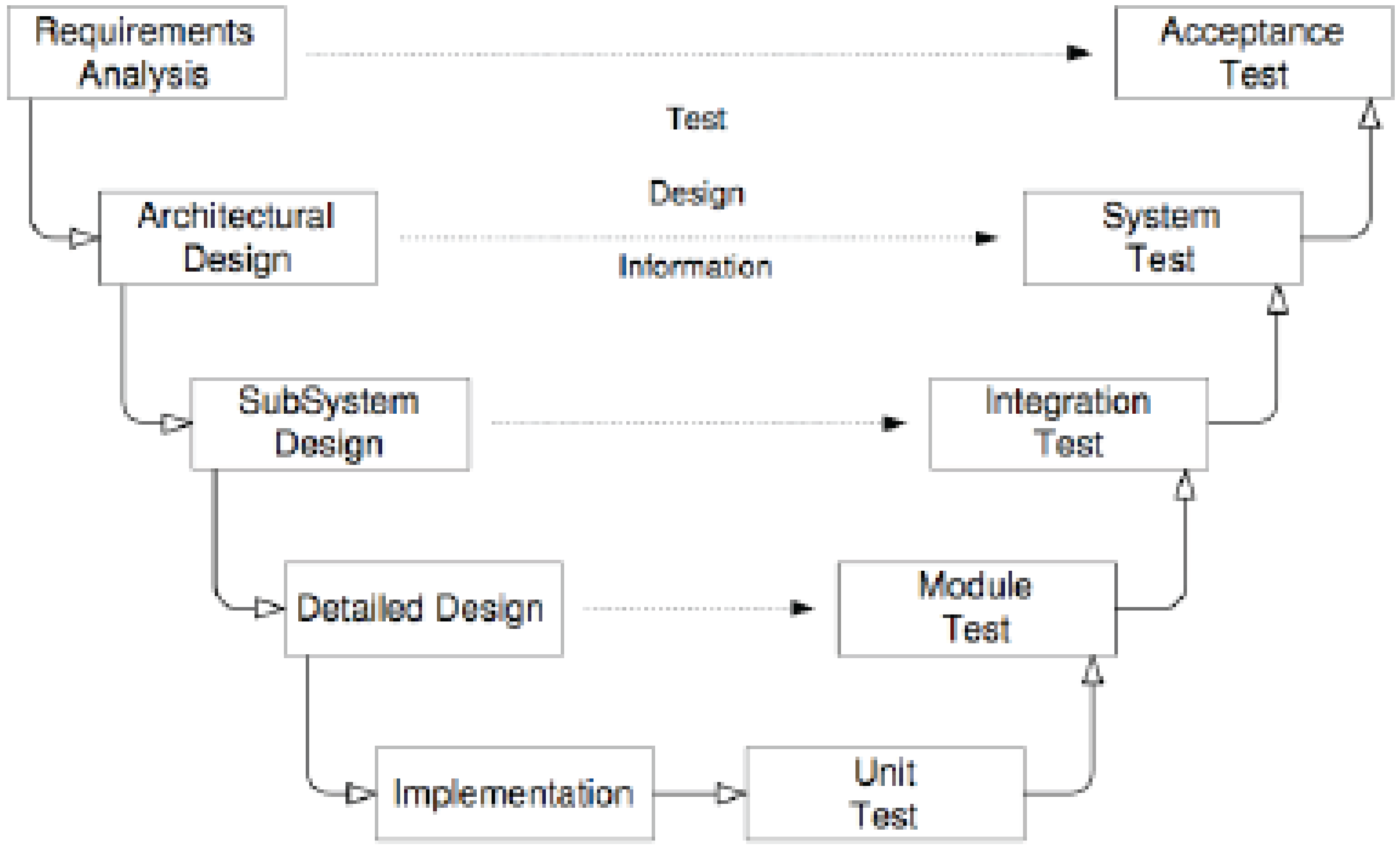
# Testing: Concepts

- Test suite
  - A collection of test cases
  - Usually these test cases share similar pre-conditions and configuration
  - Usually can be run together in sequence
  - Different test suites for different purposes
    - Smoke test, Certain platforms, Certain feature, performance, …
- Test Script
  - A script to run a sequence of test cases or a test suite automatically

## Testing: Concepts

- Test Driver
  - A software framework that can load a collection of test cases or a test suite
  - It can usually handle the configuration and comparison between expected outputs and actual outputs
- Test Coverage
  - A measurement to evaluate how well the testing is done
  - The measure can be based on multiple elements
    - Code
    - Input combinations
    - Specifications

# Granularity of Testing: V-model



## Granularity of testing

- Unit / Module Testing
  - Test of a single module
- Integration Testing
  - Test the interaction between modules
- System Testing
  - Test the system as a whole, by developers on test cases
- Acceptance Testing
  - Validate the system against user requirements, by customers with no formal test cases

# Stage of Software Testing

- Development-time testing
  - Unit testing, Integration Testing
- Before-release testing
  - System testing, Acceptance Testing
- User testing
  - Actual usage -> field bugs & patches

## Continuous Integration/Delivery

- Teams integrate their work multiple times per day.
- Each integration is verified by an automated build
- Significantly reduces integration problems
- Develop cohesive software more rapidly
- Deliver new features gradually for users to get use to them

## Types of testing by how they are designed

- Black box testing
  - The tester are just like normal users
  - They just try to cover input space and corner cases
- White box testing
  - The tester knows everything about the implementation
  - They knows where the bugs are more probably be
  - They can exercise paths in the code

## Black Box Testing: General Guidelines

- Divide value range and cover each part
  - Cover boundary values
  - Try to reach all error messages
  - Try to trigger potential exceptions
  - Feed invalid inputs
    - wrong formats, too long, too short, empty, …
  - Try combinations of all above
- Repeat same and use different inputs for many times if the input is a sequence

## White Box Testing: General Guidelines

- Try to cover all branches
  - Study the relationship between input value and branch logic
- Test more on complex modules
  - Measure complexities of modules by code size, number of branches and loops, number of calls and recursions

## White Box Testing: Techniques

- More difficult than black box testing
- Need to understand the code
- Code Coverage Guided
- Automatic support
  - Symbolic execution
  - Complexity measurement and Defect prediction

# Test Coverage

- After we have done some testing, how do we know the testing is enough?
- The most straightforward: input coverage
- # of inputs tested / # of possible inputs

- Unfortunately, # of possible inputs is typically infinite
- Not feasible, so we need approximations…

## Code Coverage

- Basic idea:
  - Bugs in the code that has never been executed will not be exposed
  - So the test suite is definitely not sufficient
- Definition:
  - Divide the code to elements
  - Calculate the proportion of elements that are executed by the test suite

## Code Coverage

- Criteria
  - Statement (basic block) coverage, are they the same?
  - Branch coverage (cover all edges in a control flow graph), same with basic block coverage?
  - Data flow coverage
  - Class/Method coverage

# Code Coverage

- Criteria
  - Statement (basic block) coverage, are they the same?
  - Branch coverage (cover all edges in a control flow graph), same with basic block coverage?
  - Data flow coverage
  - Class/Method coverage

## Thank You !

Questions ?