CS1632:
Unit Testing, part 1

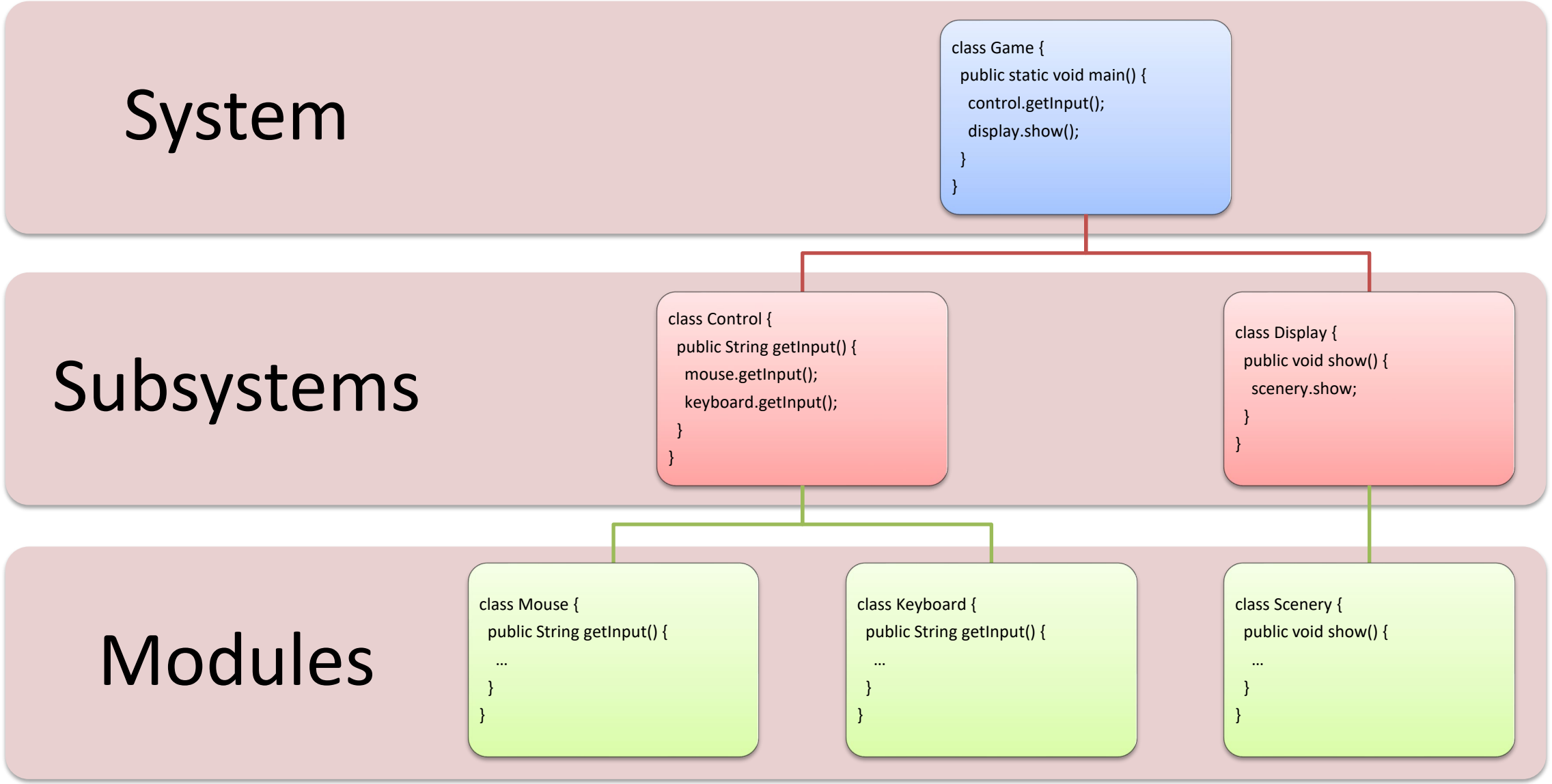
Wonsun Ahn

What is unit testing?

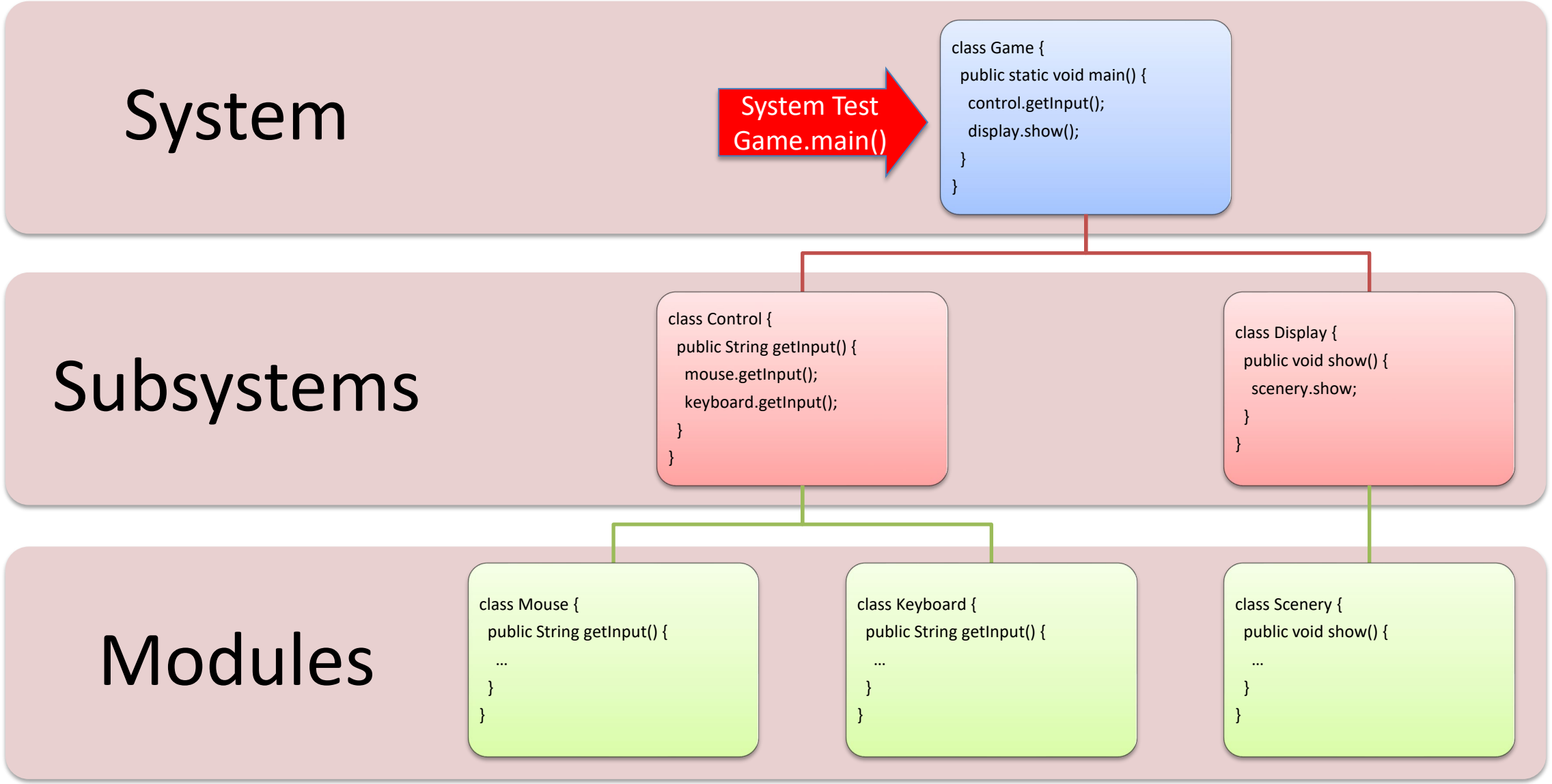
- **Unit testing**: testing small "units" of code instead of whole system
  - Units can be subsystems, modules, all the way down to individual methods
  - Most commonly refers to testing methods by directly invoking them
  - White-box testing, typically automated by a testing script

- Goal: Ensure unit works independent of rest of the system
  - Does NOT ensure that units work together well when integrated
  - Need **integration testing** for that purpose

Why Unit Test?



System Test tests Everything. What's the point?



What if System Test Fails? Where's the Bug?

System

Subsystems

Modules

System Test
Game.main()

```
class Game {
  public static void main() {
    control.getInput();
    display.show();
  }
}
```

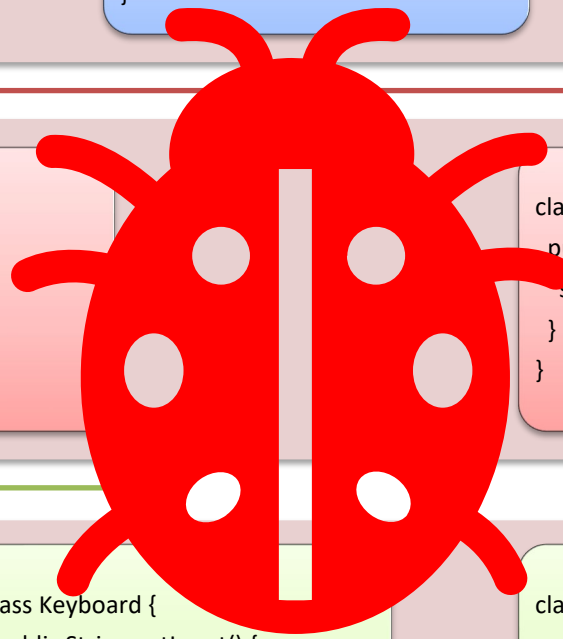
```
class Control {
  public String getInput() {
    mouse.getInput();
    keyboard.getInput();
  }
}
```

```
class Display {
  public void show() {
    scenery.show;
  }
}
```

```
class Mouse {
  public String getInput() {
    …
  }
}
```

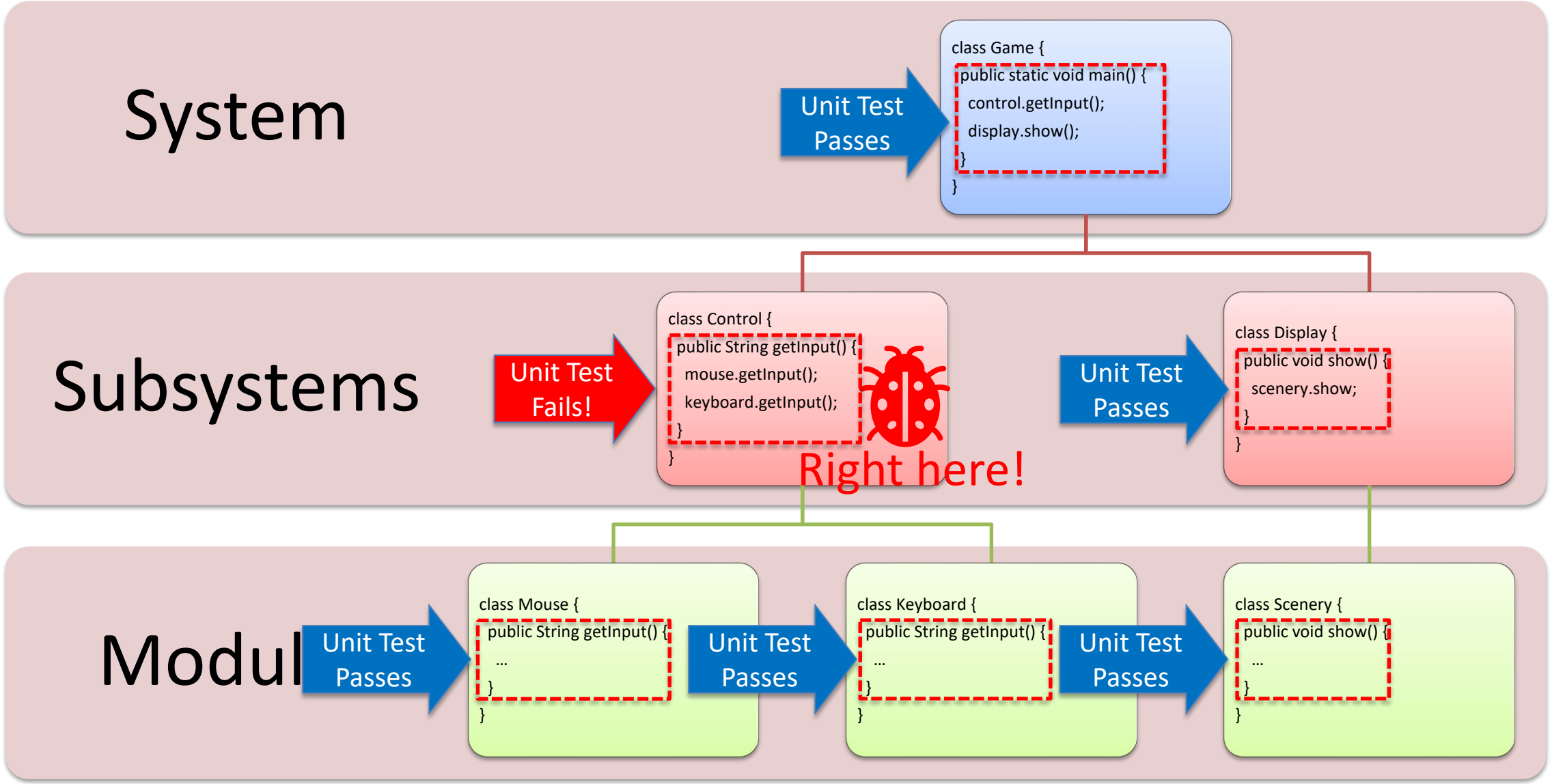
```
class Keyboard {
  public String getInput() {
    …
  }
}
```

```
class Scenery {
  public void show() {
    …
  }
}
```

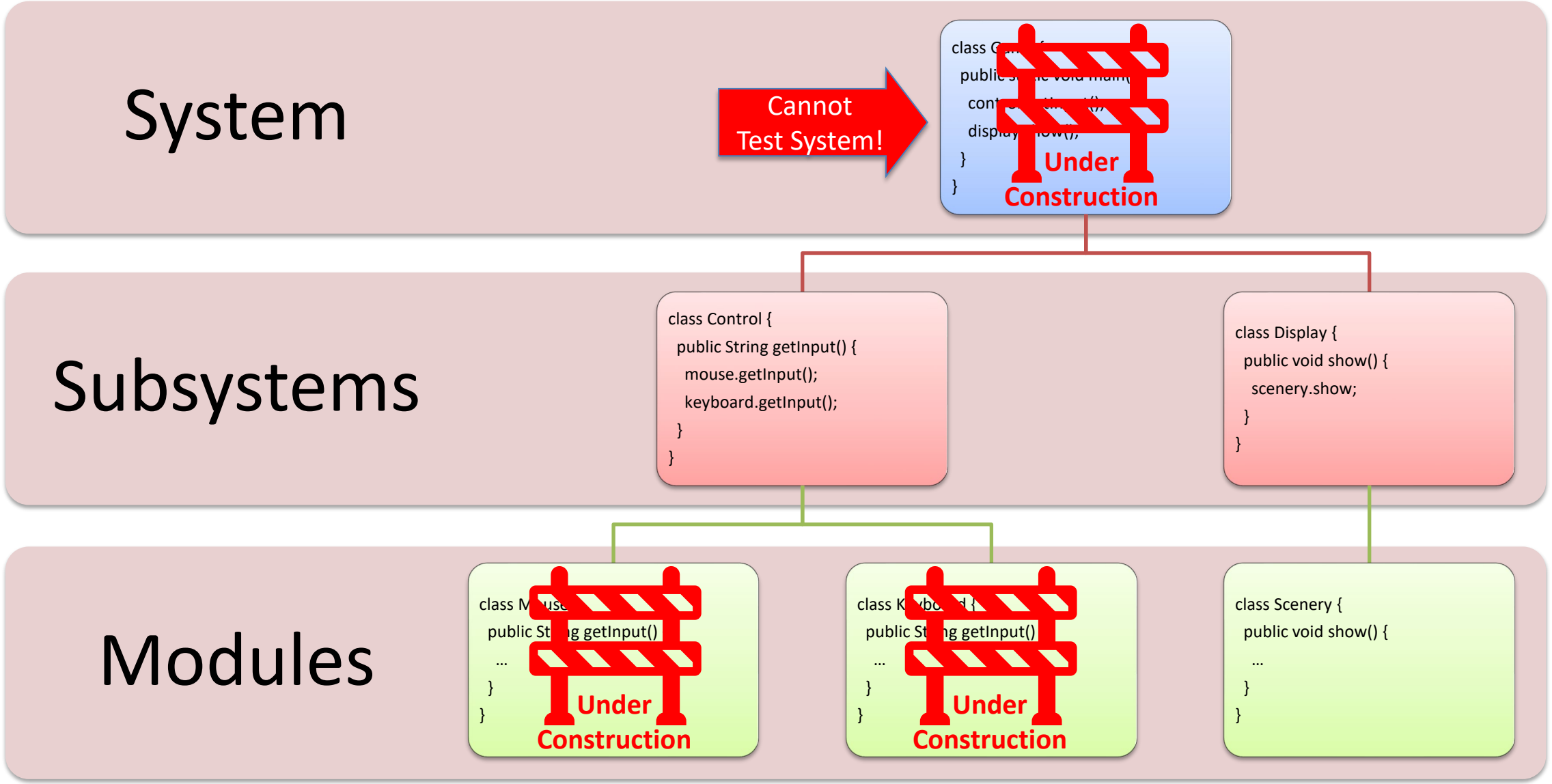


Could be anywhere!

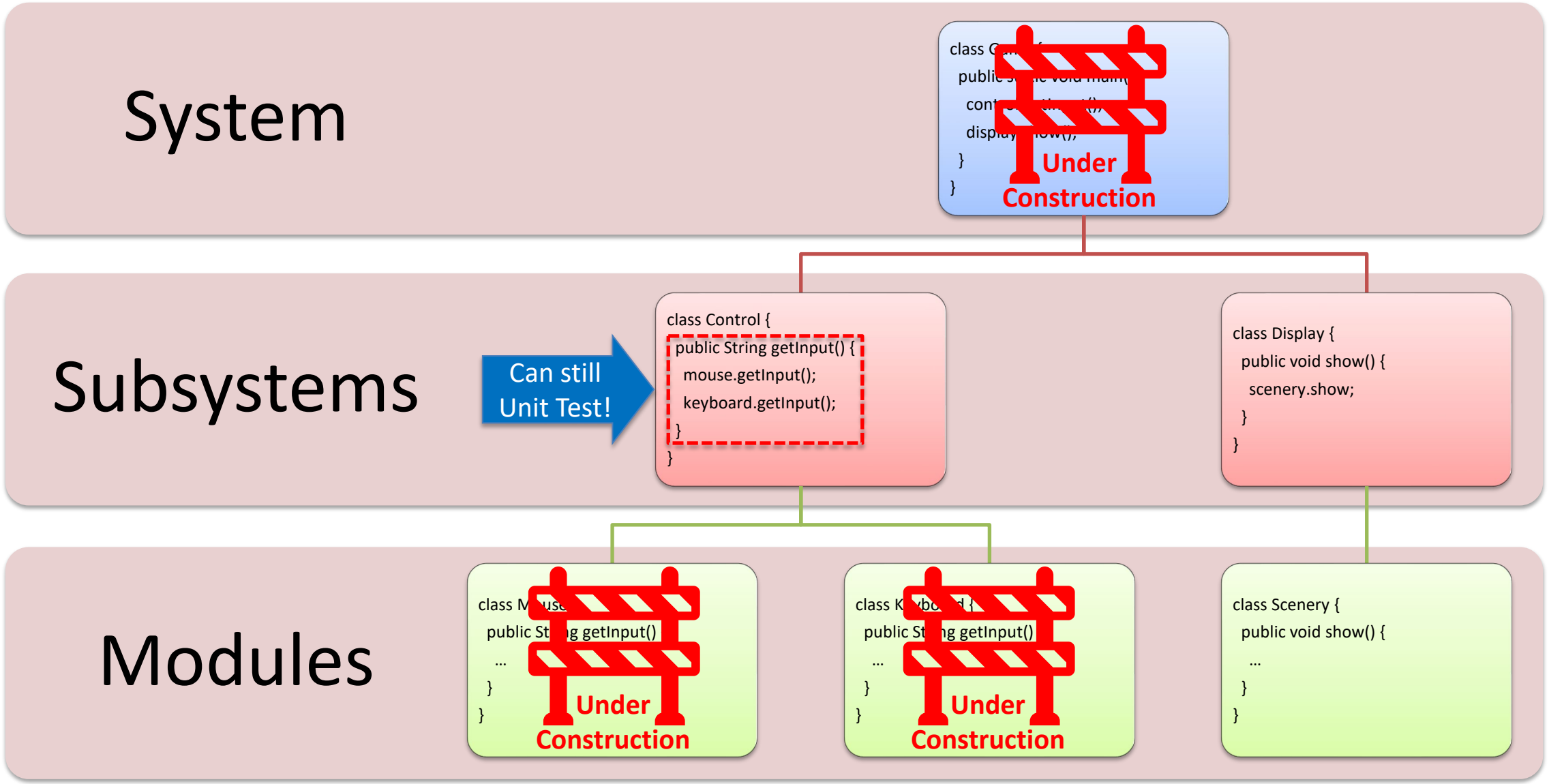
1. Unit Testing Localizes the Bug



What if System is Being Built? Can it be Tested?



2. Unit Testing Allows Testing Early On



Unit Testing is Done by Developers

Unit Test Code

```
class ControlTest {
  @Test
  public void testGetInput() {
    String str = control.getInput();
    // Do postcondition checks on str
  }
}
```

Unit Implementation Code

```
class Control {
  public String getInput() {
    String str = mouse.getInput();
    str += keyboard.getInput();
    return str;
  }
}
```

- Unit test code is developed in concert with implementation code
  - In Test Driven Development (TDD), test code is written before implementation
- Developers know best about the behavior of individual methods
- Allows immediate testing without waiting for other units to complete

Why do Unit Testing?

1. Can localize defects to a small unit of code
   - Easier to locate bug compared having to scan entire code base

2. Can perform testing early on during development (a.k.a. **shift left**)
   - Unit tests can be made into a regression test suite with good coverage

3. Unit tests serve as “living documentation”
   - Unit tests can be viewed as a documentation of expected behavior
   - Documentation is living because tests will fail if they become stale

JUnit Framework

- **JUnit**: A framework for automated unit testing of Java programs

- Composed of **annotations** + **assertions**

JUnit Annotations

- Annotations are used to indicate special methods to JUnit:
  - `@Test`: A method that is run as a **test case** when JUnit is invoked
  - `@Before`: A method that sets up a common set of **preconditions** before running each test case (a.k.a. **test fixture**)
  - `@After`: A method that tears down the test fixture set up by `@Before` (if it involves external resources such as files, databases)
- A JUnit test class has multiple `@Test` methods but only one set of `@Before` and `@After` methods.
  - Typically, one JUnit test class tests all methods of an implementation class

JUnit Assertions

- Assertions are used to check **postconditions**:
  - `assertEquals, assertArrayEquals, assertSame, assertNotSame, assertTrue, assertFalse, assertNull, assertNotNull, fail(), ...`
  - `assertSame(Object expected, Object actual)`:
    Asserts that two references refer to the same object
  - `assertEquals(Object expected, Object actual)`:
    Asserts that two objects are equal
  - `fail()`:
    Always fails. Indicates code location that should not be reached.
- Refer to JUnit reference for more assertions:
  - http://junit.sourceforge.net/javadoc/org/junit/Assert.html

Example JUnit Test Class

JUnit Test Class

```
class CatTest {
  @Test void testIsRented() {
    // Precondition setup
    Cat cat = new Cat();
    cat.rent();
    // Execution step
    boolean ret = cat.isRented();
    // Postcondition check
    assertTrue(ret);
  }
  @Test void testToString() {
    Cat cat = new Cat();
    String ret = cat.toString();
    assertEquals("available cat", ret);
  }
}
```

Implementation Class

```
class Cat {
  boolean rented = false;
  public void rent() {
    rented = true;
  }
  public boolean isRented() {
    return rented;
  }
  public String toString() {
    if (rented) {
      return "rented cat";
    } else {
      return "available cat";
    }
  }
}
```

Example JUnit Test Class – Using a Test Fixture

JUnit Test Class

```
class CatTest {
  Cat cat;
  @Before void setUp() {
    // Test fixture setup
    cat = new Cat();
  }
  @Test void testIsRented() {
    cat.rent();
    boolean ret = cat.isRented();
    assertTrue(ret);
  }
  @Test void testToString() {
    String ret = cat.toString();
    assertEquals("available cat", ret);
  }
}
```

Implementation Class

```
class Cat {
  boolean rented = false;
  public void rent() {
    rented = true;
  }
  public boolean isRented() {
    return rented;
  }
  public String toString() {
    if (rented) {
      return "rented cat";
    } else {
      return "available cat";
    }
  }
}
```

Wrong JUnit Test Class – Can you see why?

JUnit Test Class

```
class CatTest {
  // Initialize cat
  Cat cat = new Cat();

  @Test void testIsRented() {
    cat.rent();
    boolean ret = cat.isRented();
    assertTrue(ret);
  }
  @Test void testToString() {
    String ret = cat.toString();
    assertEquals("available cat", ret);
  }
}
```

Implementation Class

```
class Cat {
  boolean rented = false;
  public void rent() {
    rented = true;
  }
  public boolean isRented() {
    return rented;
  }
  public String toString() {
    if (rented) {
      return "rented cat";
    } else {
      return "available cat";
    }
  }
}
```

A JUnit Test Class with Expensive Test Fixture

JUnit Test Class

```
class DatabaseTest {
  @Before void setUp() {
    populateDB(A);
  }
  @Test void test1() {
    // Reads A only.
  }
  @Test void test2() {
    // Reads A only.
  }
  @After void tearDown() {
    emptyDB(A);
  }
}
```

- Note **`@After`** is needed this time.
  - Need to empty database before repopulating for next test case.
- Having to repeatedly populate a DB is computationally expensive.
- Is there a way to optimize?

Two More JUnit Annotations

- `@BeforeClass`: Called at start of each JUnit test class
  - Used to set up part of test fixture that is computationally expensive
  - If there is a part that is read-only, wasteful to set up repeatedly

- `@AfterClass`: Called at end of each JUnit test class
  - Used to tear down part of test fixture set up in `@BeforeClass`

The JUnit Test Class – Optimized

JUnit Test Class

```
class DatabaseTest {
  @BeforeClass void setUp() {
    populateDB(A);
  }
  @Test void test1() {
    // Reads A only.
  }
  @Test void test2() {
    // Reads A only.
  }
  @AfterClass void tearDown() {
    emptyDB(A);
  }
}
```

- Note database A is read-only.
- **`@Before`** or **`@BeforeClass`** is functionally equivalent, in this case.
- But **`@BeforeClass`** is faster!

The JUnit Test Class – Wrongly Optimized

JUnit Test Class

```
class DatabaseTest {
  @BeforeClass void setUp() {
    populateDB(B);
  }
  @Test void test1() {
    // Reads and writes B.
  }
  @Test void test2() {
    // Reads and writes B.
  }
  @AfterClass void tearDown() {
    emptyDB(B);
  }
}
```

- Note database B is now read-write.
- Do you see the problem?

Using fail() to Test Exception Postcondition

JUnit Test Class

```
class CatTest {
  @Test public void testRentTwice() {
    Cat cat = new Cat();
    cat.rent();
    try {
      cat.rent();
      fail("No exception even when
renting twice in a row.");
    } catch (Exception e) {
      // Success!
    }
  }
}
```

Implementation Class

```
class Cat {
  boolean rented = false;

  public void rent() throws Exception {
    if (rented) {
      throw new Exception(
                  "already rented");
    }
    rented = true;
  }
  …
}
```

Public vs. Private Methods

- Java classes have two types of methods:
  - Public methods: comprises the public interface of the class
  - Private methods: “helper” methods used for internal implementation

- Q: Should we test private methods as well?

- Two approaches:
  - Test public methods only
  - Test every method – public and private

Argument for testing public methods only

- Private methods may be inaccessible from external test classes
  - Fortunately, Java allows access through Java reflection
- Private methods get added/removed/changed all the time
  - Because they are merely helpers and not part of the public interface
  - If we test them, we may need to modify the test code frequently
- Private methods are tested as part of public methods anyway

Private methods are tested as part of public methods

```
class Bird {
    public int fly(int n) {
      return flapLeft(n) + flapRight(n);
    }
    // Tested as part of fly call.
    private int flapLeft(int n) { … }
    private int flapRight(int n) { … }
    // Dead code! So, no need to test anyway.
    private void urinate(double f) { … }
}
```

- A test of `fly` always tests `flapLeft` and `flapRight`
- Any private method not called in `fly` is in effect *dead code*

Argument for testing every method

- Public/private distinction is arbitrary
  - They are all methods that deserve to be unit tested
- Testing private methods helps localize a bug further
  - Able to tell exactly which private method has the bug
  - If testing only public methods, can localize only up to public methods

Testing private methods helps localize a bug further

```
// Assume all the called methods are private
public boolean foo(boolean n) {
  if (bar(n) && baz(n) && beta(n)) {
    return true;
  } else if (baz(n) ^ (thud(n) || baa(n)) {
    return false;
  } else if (meow(n) || chew(n) || chirp(n)) {
    return true;
  } else {
    return false;
  }
}
```

- If `foo` fails, hard to tell which private method has the defect, or `foo` itself

So, should we test private methods or not?

- As everything in software QA, it depends on the context.
  - Depends on the complexity of the public and private methods.
  - Depends on whether you expect private methods to change often.

- If you decide to test them, here is how…

Testing private methods using Java Reflection

```
class Bird {
  private int flapLeft(int n) { … }
}
class BirdTest {
  @Test public int testFlapLeft(int n) {
    // Get method flapLeft which has one argument of int type.
    Method m = Bird.class.getDeclaredMethod("flapLeft", int.class);
    // Change method from private to public.
    m.setAccessible(true);
    // Pass arguments to invoke. 1st argument is always the instance.
    Object ret = m.invoke(new Bird(), 5);
  }
}
```

Now Please Read Textbook Chapter 13

- Read Textbook Chapter 24 for details about Java Reflection
- Also see sample_code/junit_example
  - Do “mvn test” to run all unit and integration tests
  - Or, you can open the folder in VSCode and use the Testing extension
- User manual:
  - https://junit.org/junit5/docs/current/user-guide/
- Reference Javadoc:
  - http://junit.sourceforge.net/javadoc/