CS1632:
Unit Testing, part 1

Wonsun Ahn

Unit Testing

The what and why

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

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

Subsystems

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

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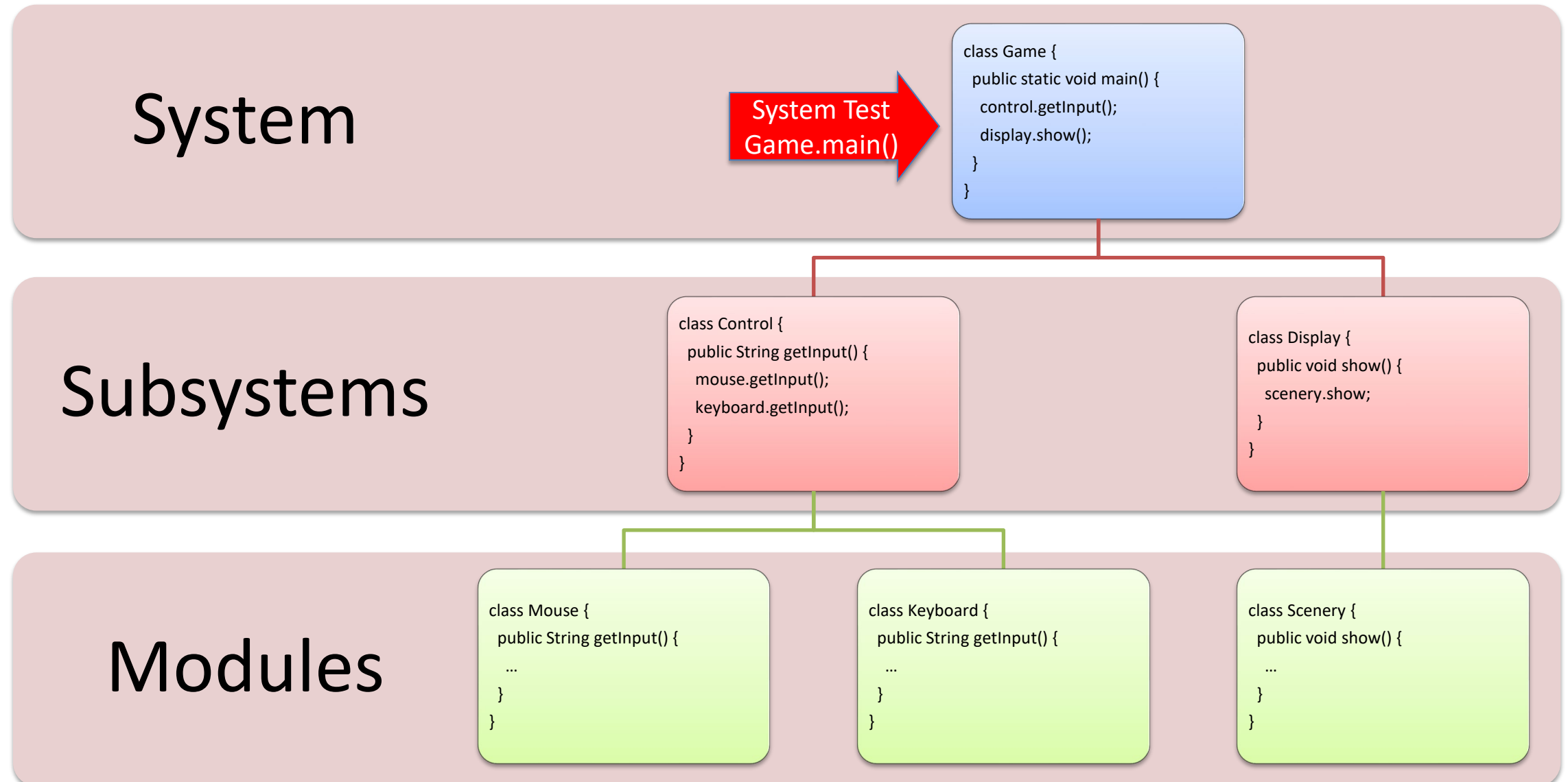
Modules

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

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

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

System Test tests Everything. What's the point?

System

System Test
Game.main()

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

Subsystems

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

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

Modules

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

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

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

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

System

System Test
Game.main()

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

Subsystems

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

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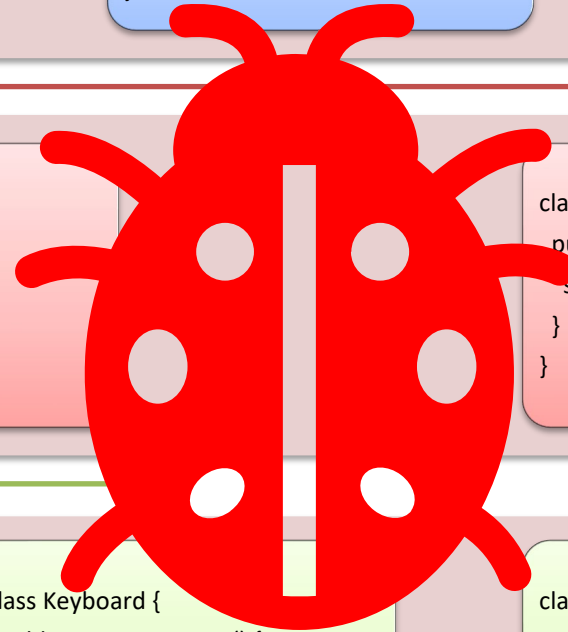
Modules

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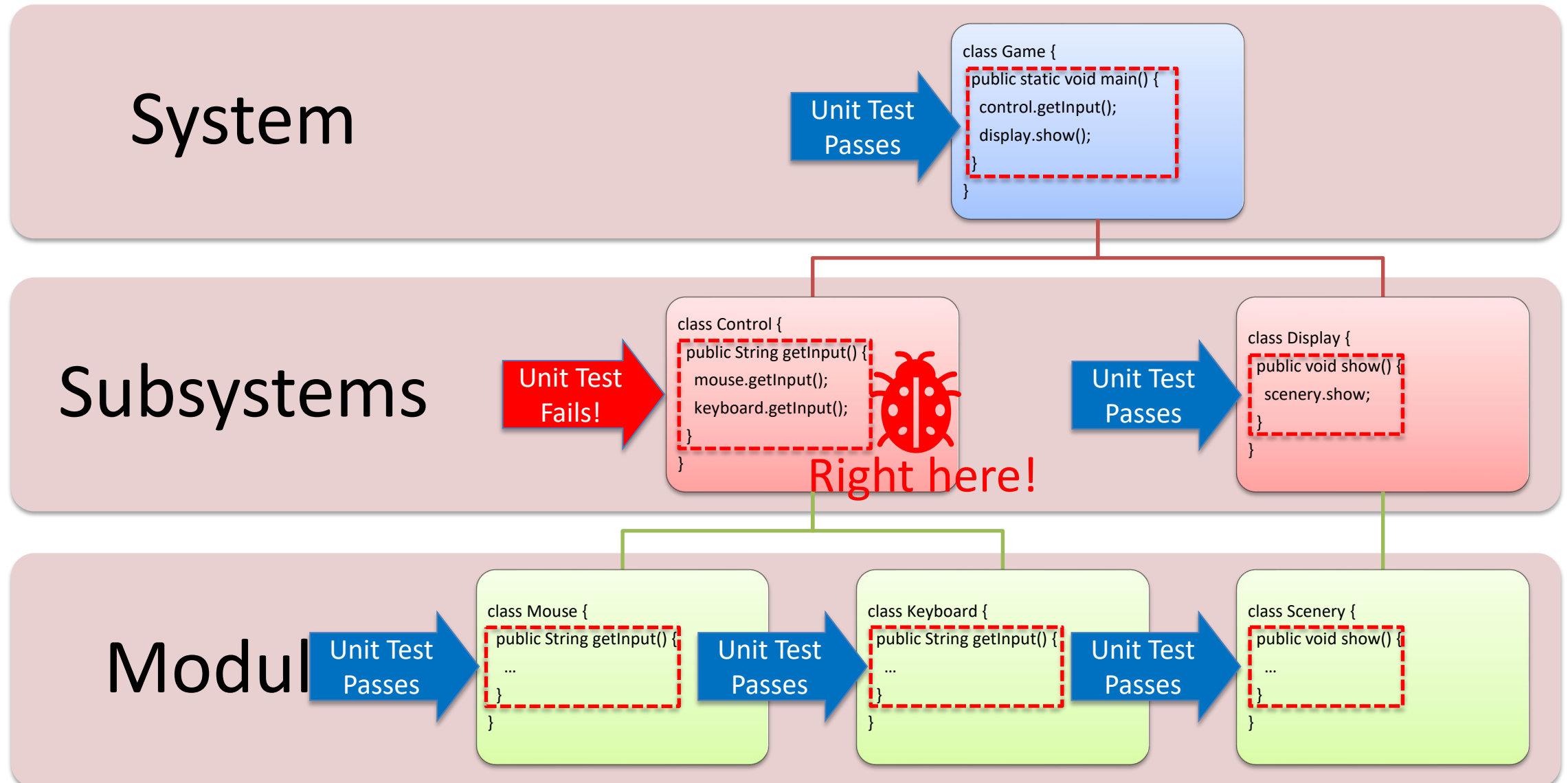
```
class Keyboard {
  public String getInput() {
  }
}
```

Could be anywhere!

```
class Scenery {
  public void show() {
  }
}
```
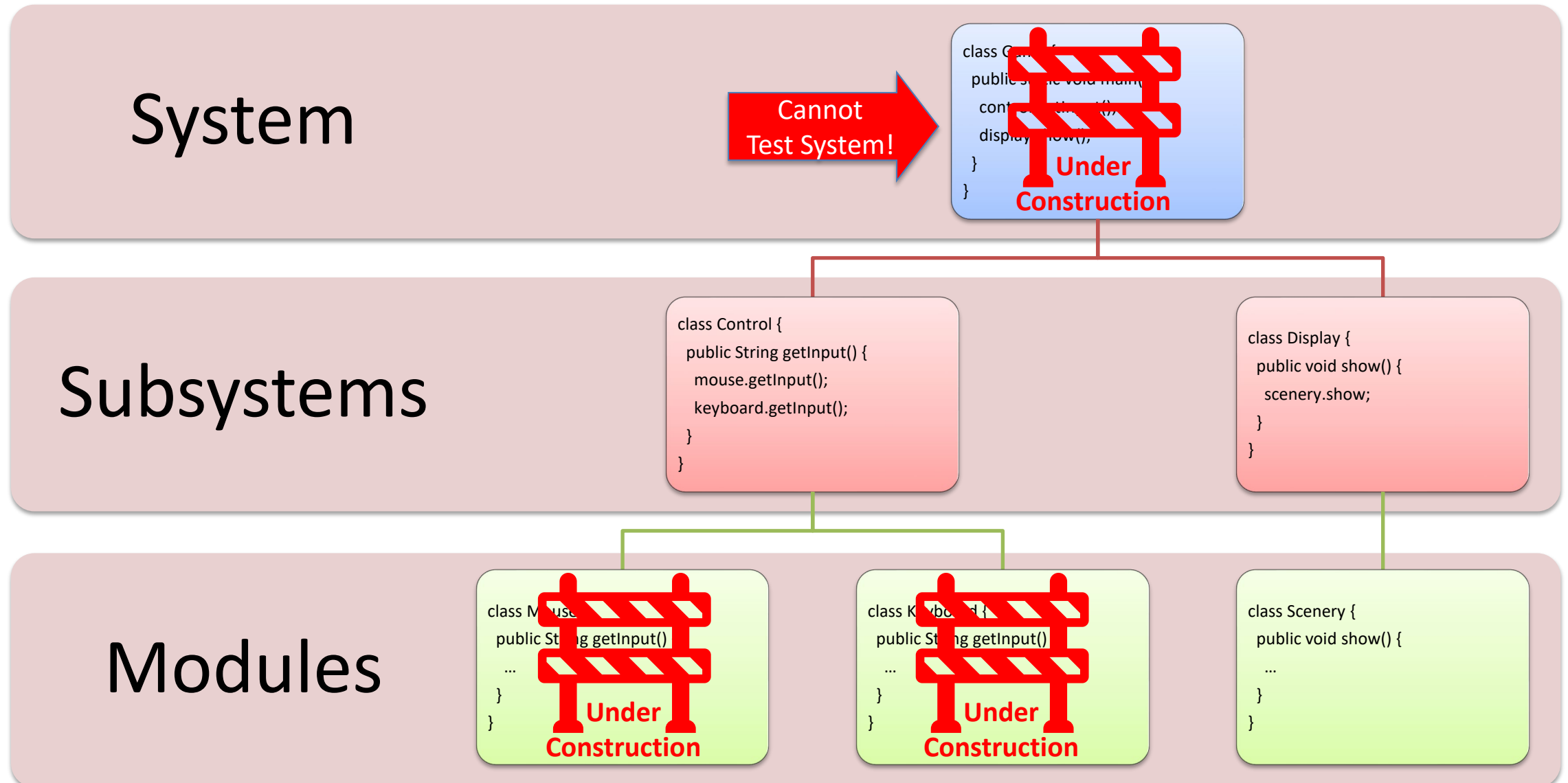
1. Unit Testing Localizes the Bug

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

System

Cannot Test System!

```
class C____ {
  public ___ void main
    cont___ __In_t()
    display__how();
  }
}
```
Under Construction

Subsystems

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

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

Modules

```
class Mouse
  public String getInput()
  ...
  }
}
```
Under Construction

```
class Keyboard {
  public String getInput()
  ...
  }
}
```
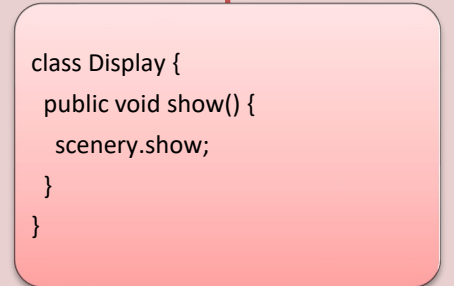**Under Construction**

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

2. Unit Testing Allows Testing Early On
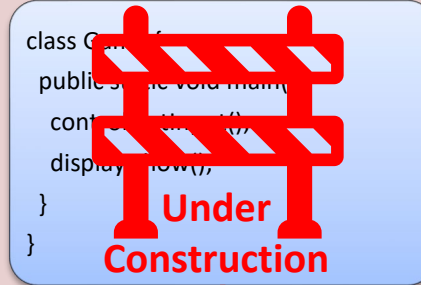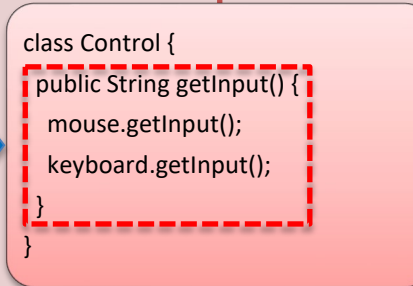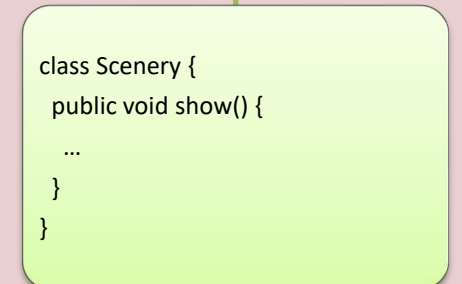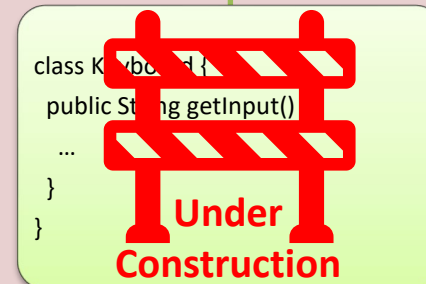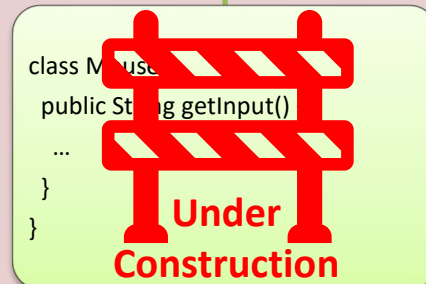
Unit Testing is Done by Developers

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**)
   - Shift-left testing makes sure technical debt does not accumulate

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

A popular framework for Java unit testing

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`: Methods run as **test cases** when JUnit test class 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 test fixture set up by `@Before` (if it needs clean up such as open files, databases)
- Typically, one JUnit test class tests one Java class
  - Consists of `@Test` methods and optionally a `@Before` and `@After`

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";
    }
  }
}
```

JUnit Assertions

- Assertions are used to check **postconditions**:
  - `assertEquals, assertArrayEquals, assertSame, assertNotSame, assertTrue, assertFalse, assertNull, assertNotNull, assertThrows, fail(), ...`
  - `assertEquals(Object expected, Object actual):`
    Asserts that two objects are equal in value.
  - `assertSame(Object expected, Object actual):`
    Asserts that two references refer to the same object.
  - `fail():`
    Always fails. Useful to indicate tests that are yet to be implemented.
- Refer to JUnit reference for more details:
  - https://junit.org/junit4/javadoc/latest/org/junit/Assert.html

JUnit is not the only unit test framework out there

- xUnit frameworks for each programming language
    - C++: CPPunit
    - JavaScript: JSUnit
    - PHP: PHPUnit
    - Python: PyUnit

- Ideas we learned apply to other testing frameworks

Unit Testing Private Methods

Should you do it? If so, how?

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...

Private methods cannot be called directly

```
class Bird {
  private int flapLeft(int n) { … }
}
class BirdTest {
  @Test public void testFlapLeft3Times() {
    // Precondition: Create a new bird.
    Bird bird = new Bird();
    // Execution Step: Flap 3 times.
    int ret = bird.flapLeft(3);   // Compiler error!
    // Postcondition: Return value is 3.
    assertEquals(3, ret);
  }
}
```

Private methods must be called via Java Reflection

```java
class BirdTest {
  @Test public void testFlapLeft3Times() throws Exception {
    // Precondition: Create a new bird.
    Bird bird = new Bird();
    // Execution Step: Flap 3 times.
    Method m = Bird.class.getDeclaredMethod("flapLeft", int.class);
    m.setAccessible(true); // Change method from private to public.
    Object ret = m.invoke(bird, 3); // Invoke flapLeft on bird.
    // Postcondition: Return value is 3.
    assertEquals(3, (int) ret);
  }
}
```

Integration Testing

Unit testing should always be followed by integration testing.

Unit Testing cannot replace Integration Testing

- A proper testing process includes both:
  - Unit tests to detect local errors within units of code
  - Integration tests to check that units work together correctly

- Units often have hidden undocumented dependencies between them
  - Since they are undocumented, they are not unit tested
  - Defects arising from these dependencies only surface when units are integrated

Hyrum's Law

"With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody."

--- Hyrum Wright

CHANGES IN VERSION 10.17:
THE CPU NO LONGER OVERHEATS WHEN YOU HOLD DOWN SPACEBAR.

COMMENTS:

LONGTIMEUSER4 WRITES:
THIS UPDATE BROKE MY WORKFLOW! MY CONTROL KEY IS HARD TO REACH, SO I HOLD SPACEBAR INSTEAD, AND I CONFIGURED EMACS TO INTERPRET A RAPID TEMPERATURE RISE AS "CONTROL".

ADMIN WRITES:
THAT'S HORRIFYING.

LONGTIMEUSER4 WRITES:
LOOK, MY SETUP WORKS FOR ME. JUST ADD AN OPTION TO REENABLE SPACEBAR HEATING.

EVERY CHANGE BREAKS SOMEONE'S WORKFLOW.

Courtesy https://www.hyrumslaw.com/

Courtesy https://xkcd.com/1172/

35

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
- JUnit 4 User Manual:
  - https://github.com/junit-team/junit4/wiki
- Junit 4 User Reference:
  - https://junit.org/junit4/javadoc/latest/