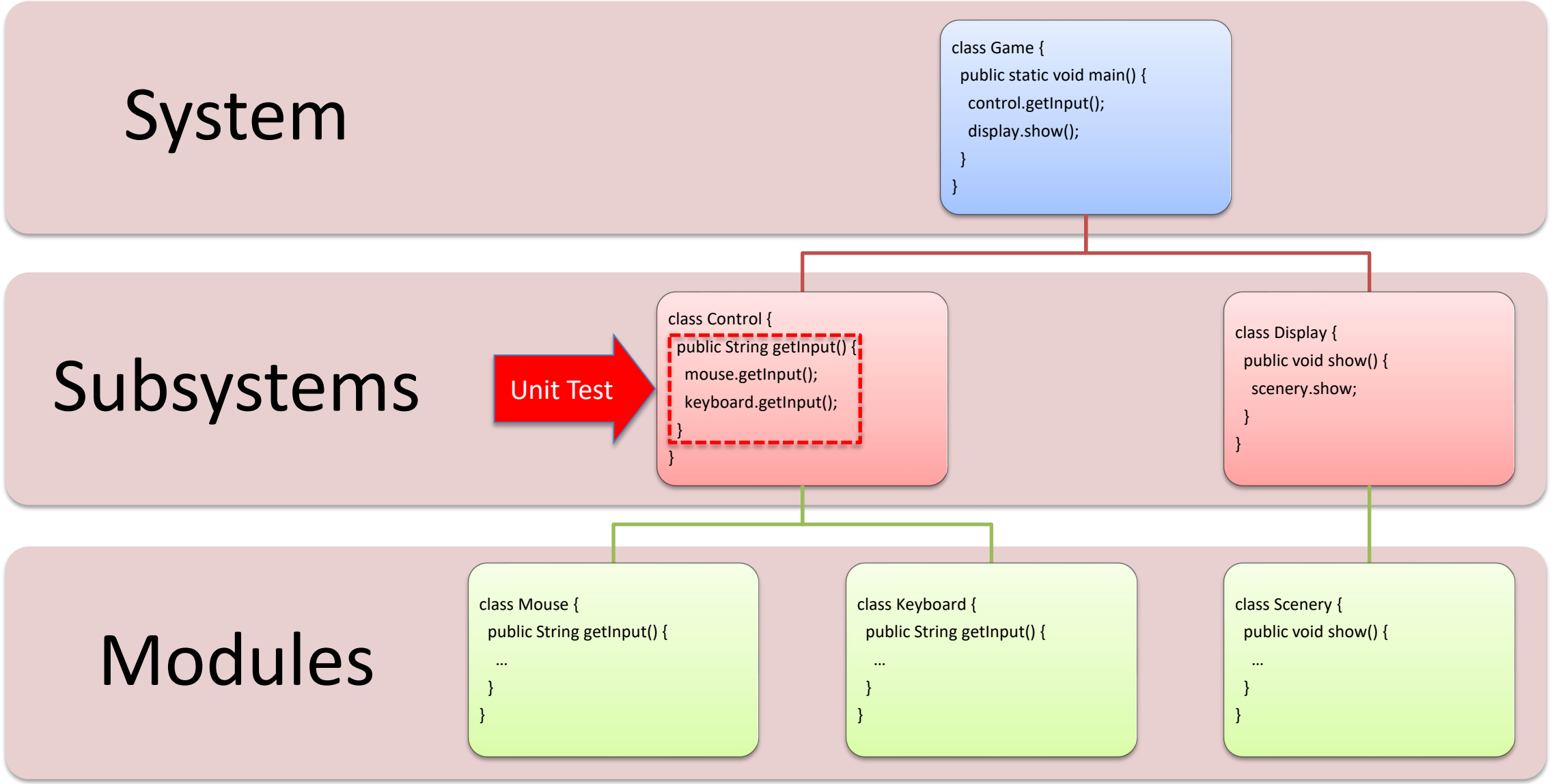
CS1632:
Unit Testing, part 2

Wonsun Ahn

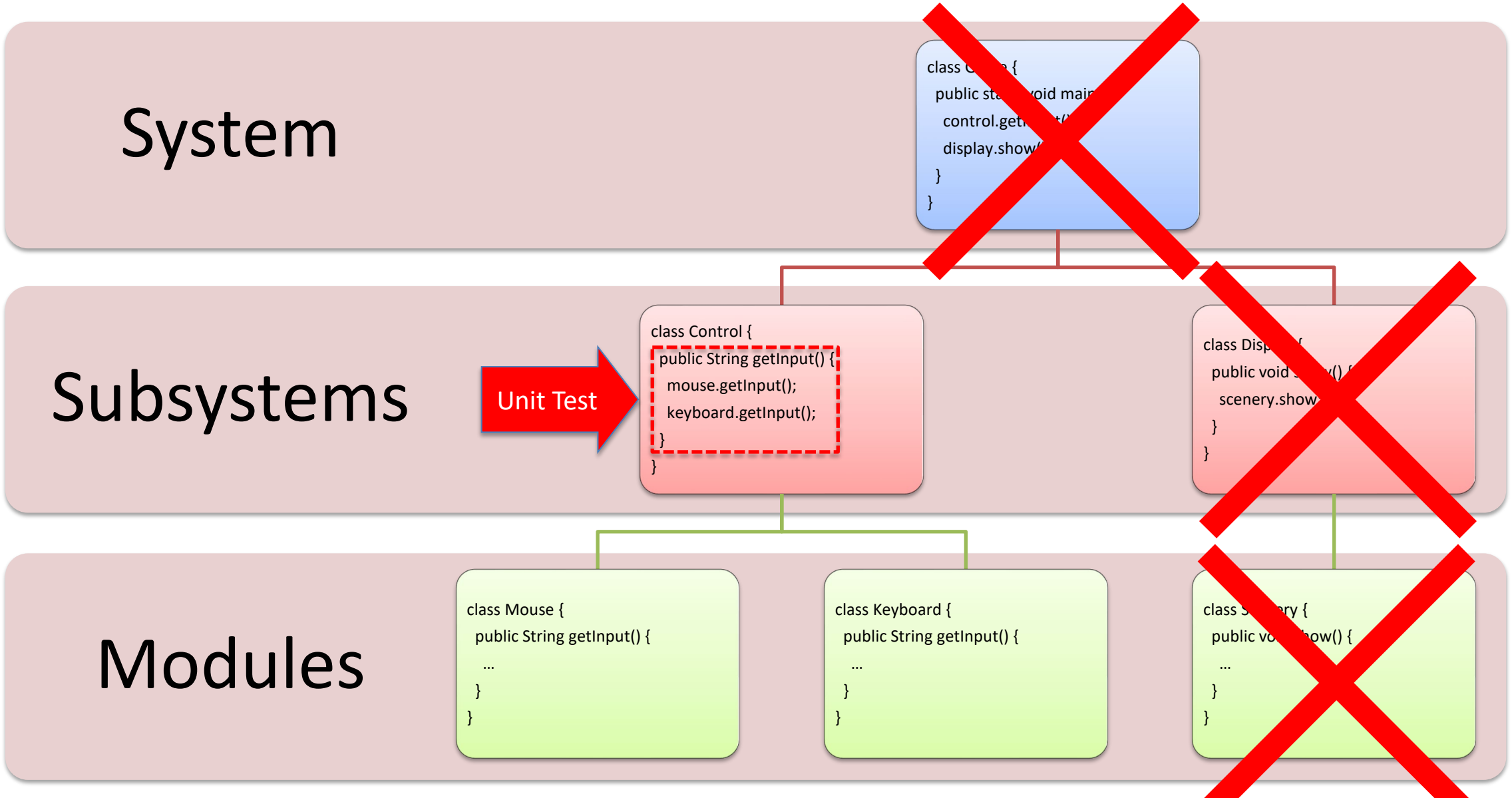
Mocks Allow True Unit Testing

By removing object dependencies from unit testing

Unit Testing Control.getInput() with Dependencies



First, let's get rid of irrelevant classes



What should we do with dependencies?

System

Subsystems

Unit Test

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

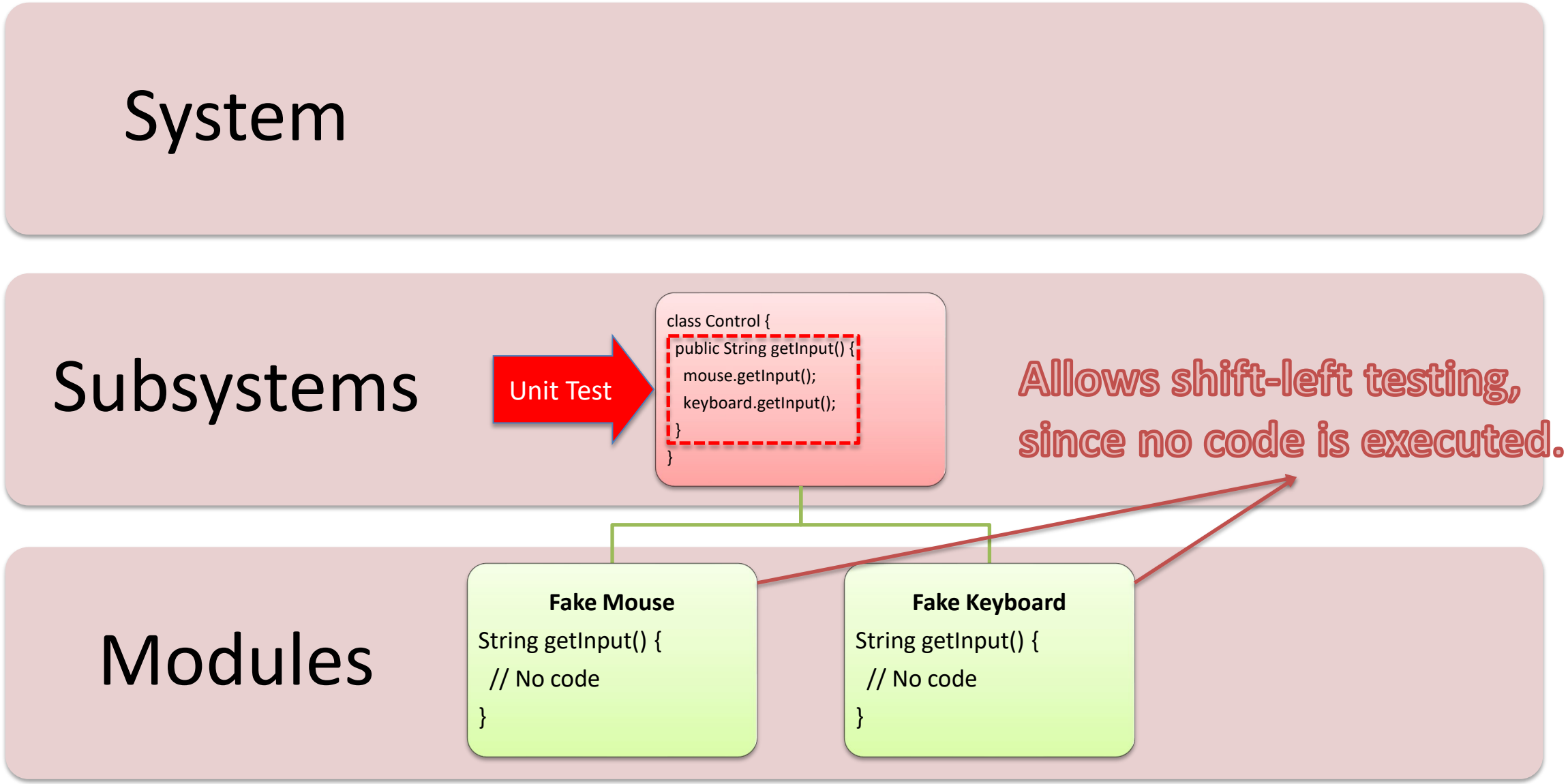
Prevent shift-left testing!

Modules

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

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

Answer: Replace dependencies with fake objects!



Test Doubles (or mocks) fake the real object

- Goal: To **not execute code** in external classes as part of the unit test.
  - Means method can be **shift-left** tested.
  - Means if a defect is found, it is **localized**.

- Just like body doubles, **test doubles** pretend to be the real thing but aren't.
  - Also called **mocks**, since they are mock-ups of the real objects.
  - Mocks appear to be real objects, but without executing any code. How???
  - Hint: mock only needs to emulate behavior for the **given test scenario.**

Running Example: Rent-A-Cat System

```
class RentACat {
  HashMap<int, Cat> cats;

  public void addCat(int id, Cat cat) {
    cats.put(id, cat);
  }
  public void rentCat(int id, int days) {
    cats.get(id).rent(days * 100);
  }
  public String listCats() {
    String ret;
    for (Cat cat : cats.values()) {
      ret += cat.toString() + "\n";
    }
    return ret;
  }
}
```

```
class Cat {
  String name;
  int netWorth = 0;

  public Cat(String name) {
    this.name = name;
  }
  public void rent(int payment) {
    netWorth += payment;
  }
  public String toString() {
    return name + " " + netWorth;
  }
}
```

RentACat depends on Cat

```
class RentACat {
  HashMap<int, Cat> cats;

  public void addCat(int id, Cat cat) {
    cats.put(id, cat);
  }
  public void rentCat(int id, int days) {
    cats.get(id).rent(days * 100);
  }
  public String listCats() {
    String ret;
    for (Cat cat : cats.values()) {
      ret += cat.toString() + "\n";
    }
    return ret;
  }
}
```

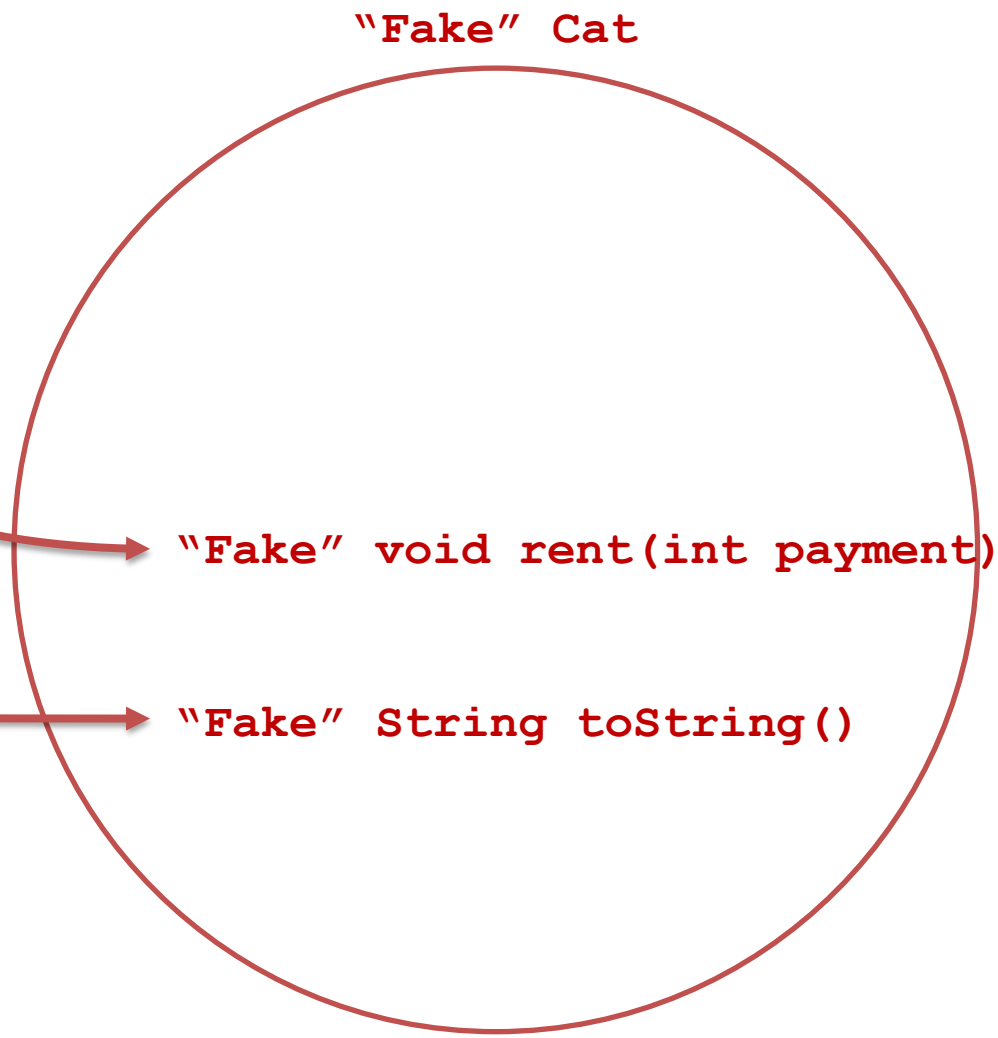
```
class Cat {
  String name;
  int netWorth = 0;

  public Cat(String name) {
    this.name = name;
  }
  public void rent(int payment) {
    netWorth += payment;
  }
  public String toString() {
    return name + " " + netWorth;
  }
}
```

How can we test RentACat w/o Cat code?

```
class RentACat {
  HashMap<int, Cat> cats;

  public void addCat(int id, Cat cat) {
    cats.put(id, cat);
  }
  public void rentCat(int id, int days) {
    cats.get(id).rent(days * 100);
  }
  public String listCats() {
    String ret;
    for (Cat cat : cats.values()) {
      ret += cat.toString() + "\n";
    }
    return ret;
  }
}
```



Mockito Framework

A popular Java framework for creating mock objects

Mock: A Fake Object with No Code

```
// Creates a mock cat of type Cat
Cat cat = Mockito.mock(Cat.class);
```

Mockito: a framework for creating test doubles

- **Mockito**: a framework for creating mocks
  - Good for emulating mocks that exhibit simple behaviors
  - Uses Java Reflection + Bytecode Rewriting to override method behavior
  - Yes, method bytecode of mocks is literally rewritten during the test!

- In Mockito terminology:
  - Test double → **Mock**, Act of creating a mock → **Mocking**

A Mock Object does not execute your code!

```
Cat cat = new Cat("Tabby");
```

```
class Cat {
  String name;
  int netWorth = 0;

  public Cat(String name) {
    this.name = name;
  }
  public void rent(int payment) {
    netWorth += payment;
  }
  public String toString() {
    return name + " " + netWorth;
  }
}
```

```
Cat cat = Mockito.mock(Cat.class);
```

```
// No member variables
// No constructor

// Default code for rent
void rent(int payment) {}
// Default code for toString
String toString() {
  return "Mock for Cat, …";
}
```

Stub: A Fake Method with No Code

```
// Rewrites toString() stub to return "Tabby"
Mockito.when(cat.toString()).thenReturn("Tabby");
```

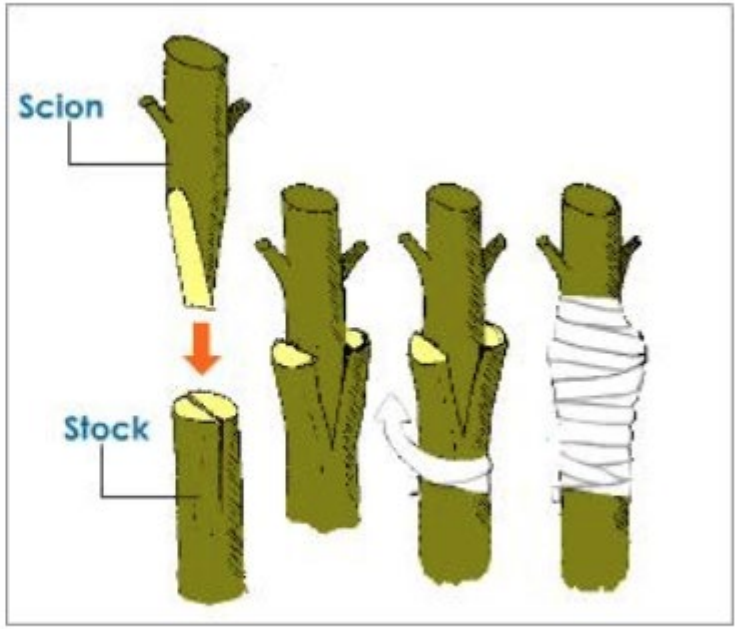
Stubbing Getter Methods Emulates State

- OOP 101: objects have *data encapsulation*
  - *Data encapsulation*: private members are not visible from outside
  - Only way to query private state is through *getter* methods
  - *Getter*: a method that returns the value of a private member variable

- What if a **precondition** specifies a state for a mock object?
  - E.g. Cat has a name of “Tabby” and a net worth of 300 dollars.
  - Answer: rewrite *getter* methods to return specified state!

Stubbing sets up preconditions.

- In Mockito terminology:
  - Fake method → **Stub**, Act of changing method return value → **Stubbing**



Courtesy: Bainbridge Island Fruit Club

- Grafts apple tree limb to the stub of another tree.
- For all purposes, tree acts like an apple tree!
  - If precondition says red apples, stub red apples
  - If precondition says green apples, stub green apples

Creating a cat named “Tabby”

```
Cat cat = new Cat(“Tabby”);
```

```
class Cat {
  String name;
  int netWorth = 0;

  public Cat(String name) {
    this.name = name;
  }
  public void rent(int payment) {
    netWorth += payment;
  }
  public String toString() {
    return name + “ “ + netWorth;
  }
}
```

```
Cat cat = Mockito.mock(Cat.class);
Mockito.when(cat.toString()).thenReturn(“Tabby 0”);
```

```
void rent(int payment) {}

// Now returns “Tabby 0”!
String toString() {
  return “Mock for Cat, …”;
  return “Tabby 0”;
}
```

Creating a cat named “Tabby” with net worth 5

```
Cat cat = new Cat(“Tabby”);
cat.rent(5);
```

```
Cat cat = Mockito.mock(Cat.class);
Mockito.when(cat.toString()).thenReturn(“Tabby 5”);
```

```
class Cat {
  String name;
  int netWorth = 0;

  public Cat(String name) {
    this.name = name;
  }
  public void rent(int payment) {
    netWorth += payment;
  }
  public String toString() {
    return name + “ “ + netWorth;
  }
}
```

```
void rent(int payment) {}

// Now returns “Tabby 5”!
String toString() {
  return “Mock for Cat, …”;
  return “Tabby 5”;
}
```

Integration Testing listCats()

```
class IntegrationTest {
  @Test
  public void testListCats() {
    // Preconditions: System has a cat named "Tabby", net worth 300, ID 1.
    RentACat rentACat = new RentACat();
    Cat cat = new Cat("Tabby");
    rentACat.addCat(1, cat);
    rentACat.rentCat(1, 3);
    // Execution Steps: List all cats in the system.
    String str = rentACat.listCats();
    // Postconditions: "Tabby" is listed with net worth 300
    assertEquals("Tabby 300\n", str);
  }
}
```

Unit Testing listCats()

```
class UnitTest {
  @Test
  public void testListCats() {
    // Preconditions: System has a cat named "Tabby", net worth 300, ID 1.
    RentACat rentACat = new RentACat();
    Cat cat = Mockito.mock(Cat.class);
    Mockito.when(cat.toString()).thenReturn("Tabby 300");
    rentACat.addCat(1, cat);
    // Execution Steps: List all cats in the system.
    String str = rentACat.listCats();
    // Postconditions: "Tabby" is listed with net worth 300
    assertEquals("Tabby 300\n", str);
  }
}
```

Behavior Verification:
Allows postcondition checks on Mocks

```
// Verifies rent(300) has been called on cat
Mockito.verify(cat).rent(300);
```

Mock state cannot (and should not) be checked

- What if a postcondition specifies a state for a mock object?
  - E.g. Cat has net worth of 300 dollars after being rented out for 3 days.

- First Answer: Cannot be done.
  - Mock cat has no state so there is nothing to check.
  - What if we emulated the state to check through stubbing?

```
Mockito.when(cat.toString()).thenReturn("Tabby 300");
assertEquals("Tabby 300", cat.toString());
```

    This is called a *tautological test*, because it always passes regardless of defects.

- Second Answer: Should not be done.
  - You are checking something about Cat, which is beyond the scope of testing.

Modifications to Mock state *can* be checked

- What if postcondition specifies a modification to the state of a mock object?
  - E.g. Cat is given a rent payment of 300 dollars, after being rented out for 3 days.

- First Answer: Can be done.
  - Mockito framework keeps track of all calls to mock objects.
  - Can check that rent call has been made (once) with a certain payment argument:

```
Mockito.verify(cat).rent(payment);
Mockito.verify(cat, Mockito.times(1)).rent(payment);
```

- Second Answer: Should be done.
  - You are checking something about RentACat, that it initiates the modification.

Setter methods are targets of behavior verification

```
class RentACat {
  HashMap<int, Cat> cats;

  public void addCat(int id, Cat cat) {
    cats.put(id, cat);
  }
  public void rentCat(int id, int days) {
    cats.get(id).rent(days * 100);
  }
  public String listCats() {
    String ret;
    for (Cat cat : cats.values()) {
      ret += cat.toString() + "\n";
    }
```

```
// No state to check

// Just stubs (no code)
void rent(int payment) {}

String toString() {
  return <stubbed value>;
}
```

Mock Cat has no state to verify.
Instead, check that RentACat correctly pays the Cat.

Getter methods are not targets of verification

```
class RentACat {
  HashMap<int, Cat> cats;

  public void addCat(int id, Cat cat) {
    cats.put(id, cat);
  }
  public void rentCat(int id, int days) {
    cats.get(id).rent(days * 100);
  }
  public String listCats() {
    String ret;
    for (Cat cat : cats.values()) {
      ret += cat.toString() + "\n";
    }
    return ret;
```

Verify?

```
// No state to check

// Just stubs (no code)
void rent(int payment) {}

String toString() {
  return <stubbed value>;
}
```

Testing is checking observed behavior == expected behavior.
Calling toString() doesn't result in changes to observed state.

Getter methods are not targets of verification

```
class RentACat {
  …
  // New version of listCats()
  public String listCats() {
    String ret;
    for (Cat cat : cats.values()) {
      ret += cat.getName() + " " +
             cat.getNetWorth() + "\n";
    }
    return ret;
  }
}
```

```
// New version of Cat
void rent(int payment) {…}
String toString() {…}
String getName() {…}
int getNetWorth() {…}
```

Verifying toString() fails even when RentACat behavior is same.

Integration Testing rentCat()

```
class IntegrationTest {
  @Test
  public void testRentCat() {
    // Preconditions: System has cat named "Tabby", net worth 0, ID 1.
    RentACat rentACat = new RentACat();
    Cat cat = new Cat("Tabby");
    rentACat.addCat(1, cat);
    // Execution Steps: Rent out "Tabby" for 3 days (100 USD / day).
    rentACat.rentCat(1, 3);
    // Postconditions: "Tabby" has net worth 300
    assertEquals("Tabby 300\n", rentACat.listCats());
  }
}
```

Unit Testing rentCat()

```
class IntegrationTest {
  @Test
  public void testRentCat() {
    // Preconditions: System has cat named "Tabby", net worth 0, ID 1.
    RentACat rentACat = new RentACat();
    Cat cat = Mockito.mock(Cat.class);
    Mockito.when(cat.toString()).thenReturn("Tabby 0");
    rentACat.addCat(1, cat);
    // Execution Steps: Rent out "Tabby" for 3 days (100 USD / day).
    rentACat.rentCat(1, 3);
    // Postconditions: "Tabby" is given payment of 300
    Mockito.verify(cat).rent(300);
}
```

Pitfall: Using Verify on a Getter Method

```
class UnitTest {
  @Test
  public void testListCats() {
    // Preconditions: System has cat named "Tabby", net worth 0, ID 1.
    RentACat rentACat = new RentACat();    Cat cat = Mockito.mock(Cat.class);
    Mockito.when(cat.toString()).thenReturn("Tabby 300");
    rentACat.addCat(1, cat);
    // Execution Steps: List all cats in the system.
    String str = rentACat.listCats();
    // Postconditions: The list consists of "Tabby" with net worth 300.
    Mockito.verify(cat).toString(); // Pointless. Nothing to do with outcome.
    assertEquals("Tabby 300\n", str); // This is what you should be testing!
  }
}
```

Pitfall: Using Mockito API on Real Objects

- Mockito.when and Mockito.verify only work on stubs of mocks.
  - Only stubs can be rewritten by Mockito to emulate or verify behavior.
  - Real methods cannot be rewritten by Mockito.

- And why would you use them on real methods to begin with?
  - Real Object == Tested Object, Real Method == Tested Method.
  - Using Mockito.when to stub behavior of tested method doesn’t make sense.
  - Using Mockito.verify to verify tested method is called doesn’t make sense.

Pitfall: Using Mockito API on Real Objects

- How about real methods that are not tested methods?
  - Tested method often calls private “helper” methods within tested object.

- Helper methods are considered to be part of unit test. Rationale:
  1. This does not prevent shift-left testing.
     (Helpers are part of the tested class that is being currently developed.)
  2. There is no good way to fake helper methods within same object.
     (Unlike external objects which leveraged data encapsulation of OOP.)

Mocking has Uses Other than Unit Testing

- Robustness testing: for emulating hardware device failures
  - Hard to induce failures in real devices such as hard disks
  - Emulate failure in mock device to test how the system responds

- Repeatable testing: for controlling random number generation
  - Hard to test programs that rely on random number generators
  - Decide exactly what numbers get generated using mock generators

Limitations of Mockito

Mockito is not best for mocking complex behavior

Now rentCat cannot be tested using mock cats

```
class RentACat {
  HashMap<int, Cat> cats;

  public void addCat(int id, Cat cat) {
    cats.put(id, cat);
  }
  // Now cat displays two different states.
  // Can't stub 2 values on cat.toString().
  public String rentCat(int id, int days) {
    Cat cat = cats.get(id).
    String ret = cat.toString() + "\n";
    cat.rent(days * 100);
    ret += cat.toString() + "\n";
    return ret;
  }
}
```

```
class Cat {
  String name;
  int netWorth = 0;

  public Cat(String name) {
    this.name = name;
  }
  public void rent(int payment) {
    netWorth += payment;
  }
  public String toString() {
    return name + " " + netWorth;
  }
}
```

Create a Fake Class when Mocking doesn't work

```
class IntegrationTest {
  @Test
  public void testRentCat3Days() {
    RentACat rentACat = new RentACat();

    Cat cat = new FakeCat3Days("Tabby");
    rentACat.addCat(1, cat);

    String str = rentACat.rentCat(1, 3);

    assertEquals("Tabby 0\nTabby 300\n", str);
  }
}
```

```
class FakeCat3Days extends Cat {
  String ret = "Tabby 0";
  int calls = 0;

  public Cat(String name) {}

  public void rent(int payment) {
    ret = "Tabby 300";
  }

  public String toString() {
    return ret;
  }
}
```

Another Fake Class for Another Test Case

```
class IntegrationTest {
  @Test
  public void testRentCat5Days() {
    RentACat rentACat = new RentACat();

    Cat cat = new FakeCat5Days("Tabby");
    rentACat.addCat(1, cat);

    String str = rentACat.rentCat(1, 5);

    assertEquals("Tabby 0\nTabby 500\n", str);
  }
}
```

```
class FakeCat5Days extends Cat {
  String ret = "Tabby 0";
  int calls = 0;

  public Cat(String name) {}

  public void rent(int payment) {
    ret = "Tabby 500";
  }

  public String toString() {
    return ret;
  }
}
```

How to Create a Fake Class

- Inherit from class you want to fake
- Override methods to remove as much code as possible
- Insert minimum amount of code to emulate correct behavior

Now Please Read Textbook Chapter 14

- Also see sample_code/junit_example
  - Do “mvn test” or use VSCode Testing extension to run tests
  - See how Node objects are mocked and stubbed in @Before setUp()
  - See how Mockito.verify is used to perform behavior verification

- Mockito User Manual:

https://javadoc.io/static/org.mockito/mockito-core/3.2.4/org/mockito/Mockito.html