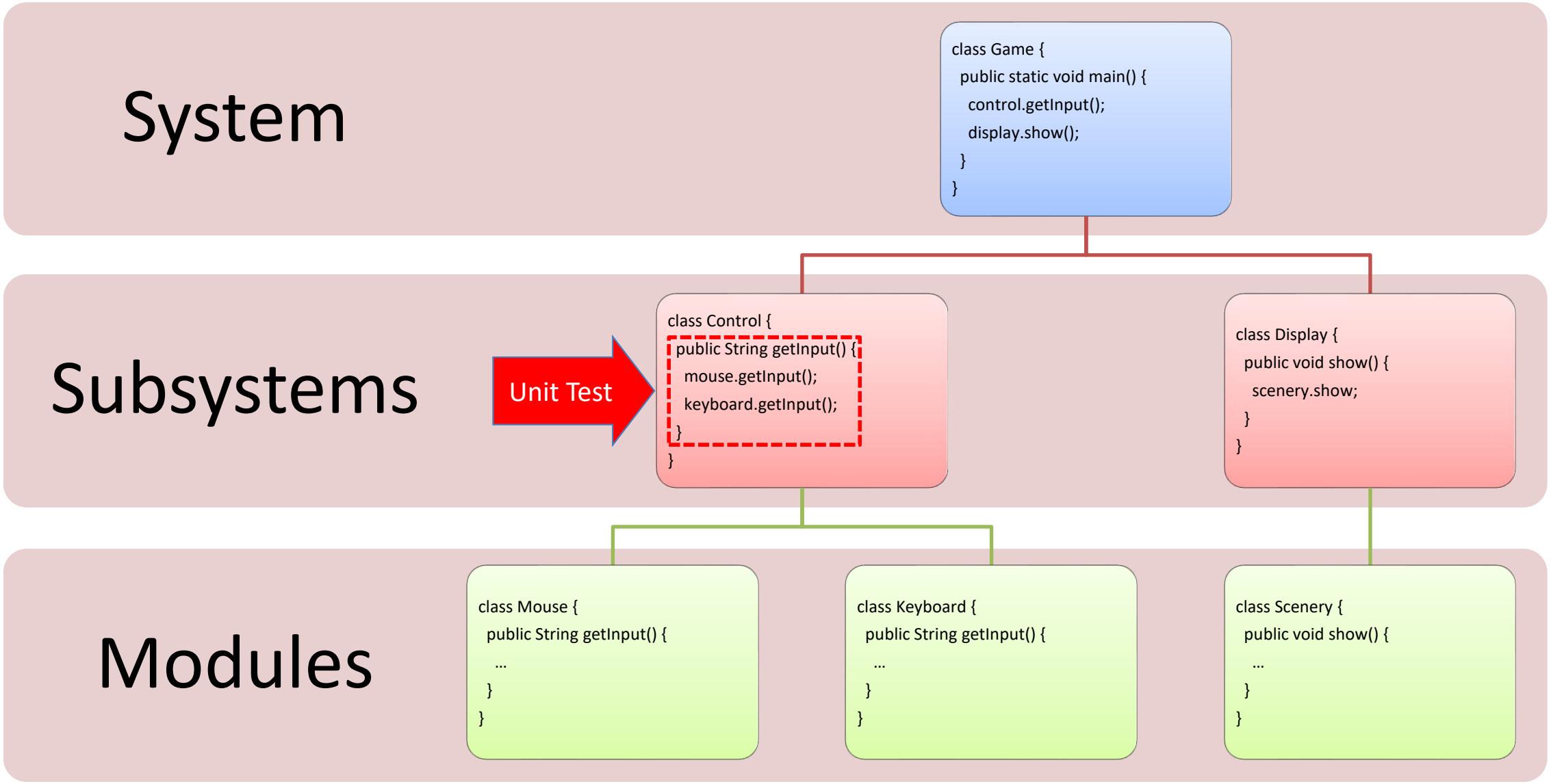
CS1632:
Unit Testing, part 2

Wonsun Ahn

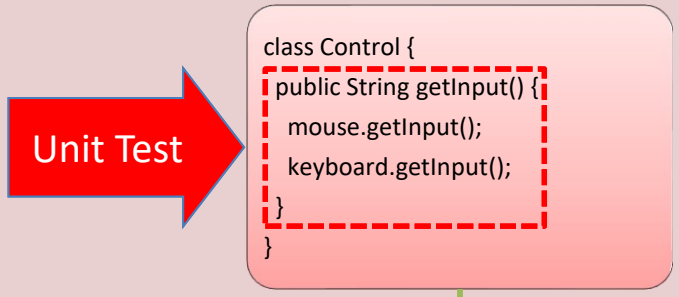
Unit Testing Control.getInput() with Dependencies



Let’s first get rid of irrelevant classes

System

Subsystems



Modules

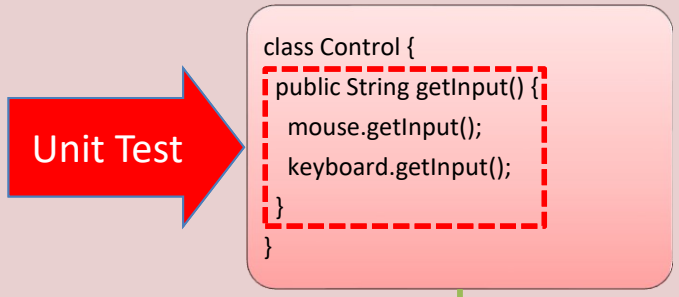
```
class Mouse {
  public String getInput() {
    …
  }
}
```

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

Unit Test using Fake Objects for Dependencies

System

Subsystems



Modules

Fake Mouse

Emulates behavior of mouse.getInput() without executing any code.

Fake Keyboard

Emulates behavior of keyboard.getInput() without executing code.

Fake Objects are called Test Doubles

- Just like body doubles, **test doubles** pretend to be the real thing, but aren't.

- Goal: To **not execute code** in external classes as part of the unit test.
  - Means if a defect is found, it is **localized** to within the tested method.
  - Means method can be tested with dependent classes still under development.

- Test double *appears* like the real thing to tested method
  - Even if double does not execute code in the external class
  - Double emulates the real object's behavior in the **given test scenario**

Running Example: Rent-A-Cat System

```
class RentACatImpl {
  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;
  }
}
```

RentACatImpl depends on Cat

```
class RentACatImpl {
  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;
  }
}
```

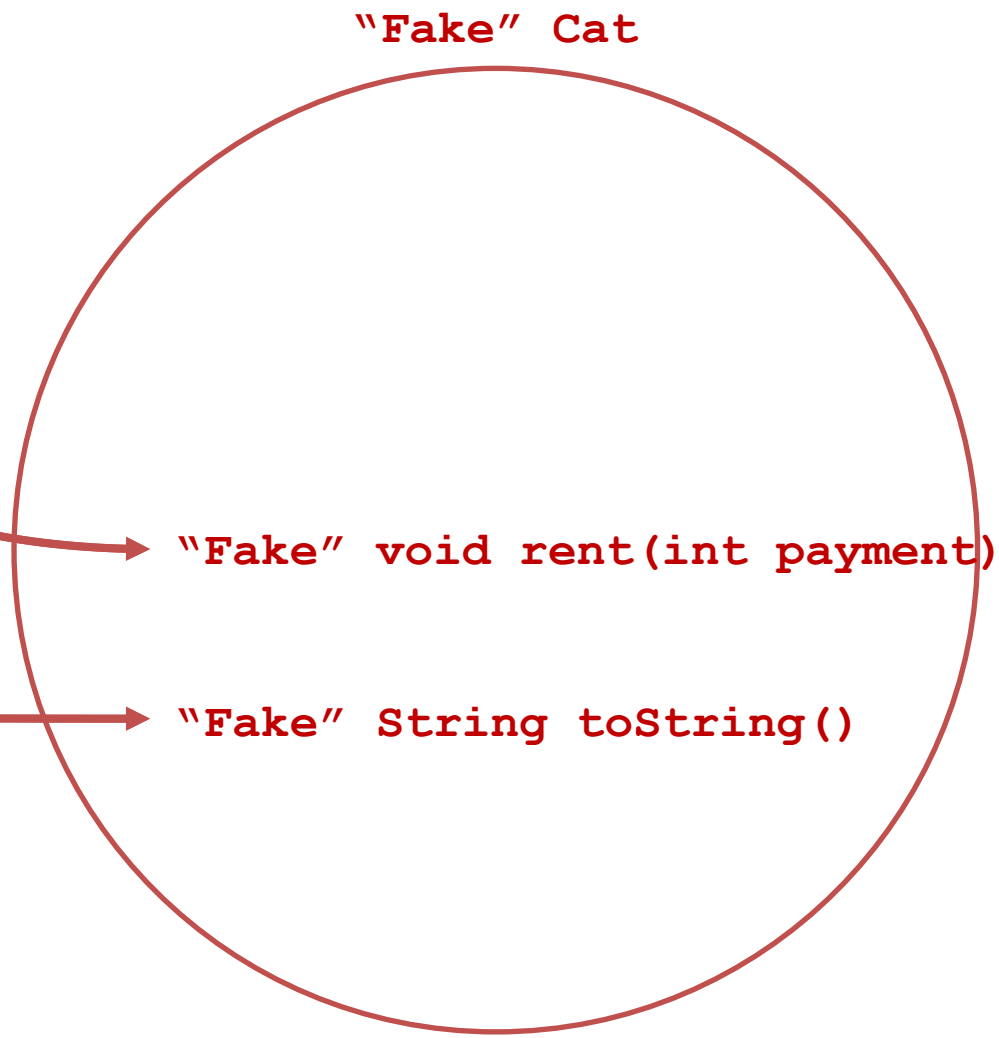
```
class Cat {
  String name;
  int netWorth = 0;

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

How can we test RentACatImpl w/o Cat code?

```
class RentACatImpl {
  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;
  }
}
```



Mocking and Stubbing

Mockito Framework

- **Mockito**: a framework for creating test doubles
  - Can emulate test doubles that exhibit simple behaviors
  - Uses Java Reflection to override method behavior

- Frequently used in conjunction with JUnit to do unit testing

- In Mockito terminology:
  - Fake object → **Mock**, Act of creating a mock → **Mocking**
  - Fake method → **Stub**, Act of updating a stub → **Stubbing**

A Mock Object contains No 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

// Only Stubs (no code)
void rent(int payment) {}

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

Stubbing Allows Emulation of Behavior

```
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");
```

```
// No Member variables

// No Constructor

// Only Stubs (no code)
void rent(int payment) {}

String toString() {
  return "Tabby 0";
}
```

Any Internal State can be Emulated

```
Cat cat = new Cat("Tabby");
cat.rent(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;
  }
}
```

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

```
// No Member variables

// No Constructor

// Only Stubs (no code)
void rent(int payment) {}

String toString() {
  return "Tabby 5";
}
```

Any Internal State can be Emulated

- True for any Java object with proper *data encapsulation*
  - Data encapsulation: when all member variables are declared private
- Only way to query internal state is through *getter* methods
  - Getter: a method that returns a value from internal state
- By stubbing getter methods, can emulate any internal state

An Integration Test using a Real Cat

```
class IntegrationTest {
  @Test
  public void testListCats() {
    RentACatImpl impl = new RentACatImpl();

    // Preconditions: Add a cat with name "Tabby" and net worth USD 300.
    Cat cat = new Cat("Tabby");
    impl.addCat(1, cat);           // Add cat with ID 1.
    impl.rentCat(1, 3);            // Rent cat ID 1 for 3 days (100 USD / day).

    String str = impl.listCats(); // Call listCats().

    assertEquals("Tabby 300\n", str);   // Depends on Cat being correct.
  }
}
```

A Unit Test using a Mock Cat

```
class UnitTest {
  @Test
  public void testListCats() {
    RentACatImpl impl = new RentACatImpl();

    // Preconditions: Add a cat with name "Tabby" and net worth USD 300.
    Cat cat = Mockito.mock(Cat.class);
    Mockito.when(cat.toString()).thenReturn("Tabby 300");
    impl.addCat(1, cat);                    // Add cat with ID 1.

    String str = impl.listCats();           // Call listCats().

    assertEquals("Tabby 300\n", str);       // Works regardless of Cat.
  }
}
```

Behavior Verification

How can we test rentCat(int id, int days)?

```
class RentACatImpl {
  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;
  }
}
```

???

Tested through stubbing

```
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;
  }
}
```

Two Types of Interaction with External Objects

```
class RentACatImpl {
  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;
  }
}
```

Calling "setter" method

Calling "getter" method

```
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;
  }
}
```

But there is no state to “set” in a Mock!

```
class RentACatImpl {
  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;
  }
}
```

```
// No Member variables

// No Constructor

// Only Stubs (no code)
void rent(int payment) {}

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

A mock has no state.
“Setter” methods are no-ops!

But there is no state to “set” in a Mock!

- Normally, you would check that rent(payment) has been called by checking state through a getter method like toString().
- But rent(payment) does nothing and toString() just returns stubbed value!
- I am stumped!

```
// No Member variables

// No Constructor

// Only Stubs (no code)
void rent(int payment) {}

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

Directly check that rent(payment) is called!

- What if there is a way to check rent(payment) is called directly?
- **State verification**: checking state
  - assertEquals, assertTrue, …
- **Behavior verification**: checking behavior (the interaction) itself
  - Mockito.verify

```
// No Member variables

// No Constructor

// Only Stubs (no code)
void rent(int payment) {}

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

An Integration Test using a Real Cat

```
class IntegrationTest {
  @Test
  public void testRentCat() {
    RentACatImpl impl = new RentACatImpl();

    // Preconditions: Add a cat with name "Tabby" and net worth USD 0.
    Cat cat = new Cat("Tabby");
    impl.addCat(1, cat);              // Add cat with ID 1.

    impl.rentCat(1, 3);               // Rent cat ID 1 for 3 days (100 USD / day).

    assertEquals("Tabby 300\n", impl.listCats());  // Verify net worth = 300
  }
}
```

A Unit Test using a Mock Cat

```
class IntegrationTest {
  @Test
  public void testRentCat() {
    RentACatImpl impl = new RentACatImpl();

    // Preconditions: Add a cat with name "Tabby" and net worth USD 0.
    Cat cat = Mockito.mock(Cat.class);
    Mockito.when(cat.toString()).thenReturn("Tabby 0");
    impl.addCat(1, cat);              // Add cat with ID 1.

    impl.rentCat(1, 3);               // Rent cat ID 1 for 3 days (100 USD / day).

    Mockito.verify(cat).rent(300);    // Verify cat.rent(300) has been called
}
```

Using Verify on a Getter is Pointless

```
class UnitTest {
  @Test
  public void testListCats() {
    RentACatImpl impl = new RentACatImpl();

    // Preconditions: Add a cat with name "Tabby" and net worth USD 300.
    Cat cat = Mockito.mock(Cat.class);
    Mockito.when(cat.toString()).thenReturn("Tabby 300");
    impl.addCat(1, cat);                    // Add cat with ID 1.

    String str = impl.listCats();           // Call listCats().

    // Passes, but what's the point?  Nothing to do with correctness.
    Mockito.verify(cat).toString();
  }
}
```

Mockito API is only usable on Mock Objects

- Mockito.when and Mockito.verify only work on mock objects.
- You should feel no need to use them on real methods to begin with.
  - Real methods = tested method + “helper” methods called by tested method
- No need to use Mockito.when (stubbing) on real methods.
  - You want to test real methods as-is. Why stub to change behavior?
- No need to use Mockito.verify (behavior verification) on real methods.
  - Whether “helper” methods are called has nothing to do with correctness.

Limitations of Mocking

Now rentCat cannot be tested using mock cats

```
class RentACatImpl {
  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";
  }
}
```

```
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() {
    RentACatImpl impl = new RentACatImpl();

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

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

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

```
class FakeCat3Days extends Cat {
  String[] arr = new String[] {
    "Tabby 0", "Tabby 300"};
  int calls = 0;

  public Cat(String name) {}

  public void rent(int payment) {}

  public String toString() {
    return arr[calls++];
  }
}
```

Another Fake Class for Another Test Case

```
class IntegrationTest {
  @Test
  public void testRentCat5Days() {
    RentACatImpl impl = new RentACatImpl();

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

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

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

```
class FakeCat5Days extends Cat {
  String[] arr = new String[] {
    "Tabby 0", "Tabby 500"};
  int calls = 0;

  public Cat(String name) {}

  public void rent(int payment) {}

  public String toString() {
    return arr[calls++];
  }
}
```

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

Discussion and Summary

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

- Reproducible 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

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 should apply to other testing frameworks easily

Unit Testing cannot replace Integration Testing

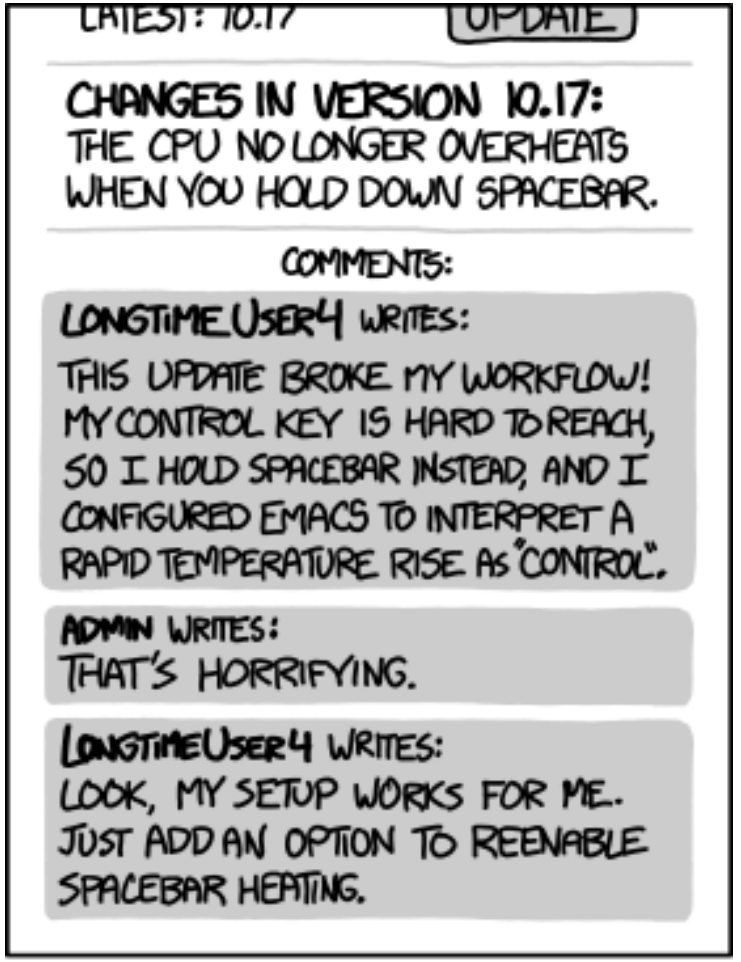
- A proper testing process includes both:
  - Unit tests to detect local errors within units of code
  - Integration tests to check that units work together correctly

- 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



Courtesy https://www.hyrumslaw.com/

Courtesy https://xkcd.com/1172/

Now Please Read Textbook Chapter 14

- Read sample_code/junit_example/LinkedListUnitTest.java
  - Replica of LinkedListTest.java we saw in Part 1 but using Mockito
  - LinkedListTest: was really an integration test, not a unit test
  - LinkedListUnitTest: true unit test that mocks all Node instances

- Mockito User Manual:

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