

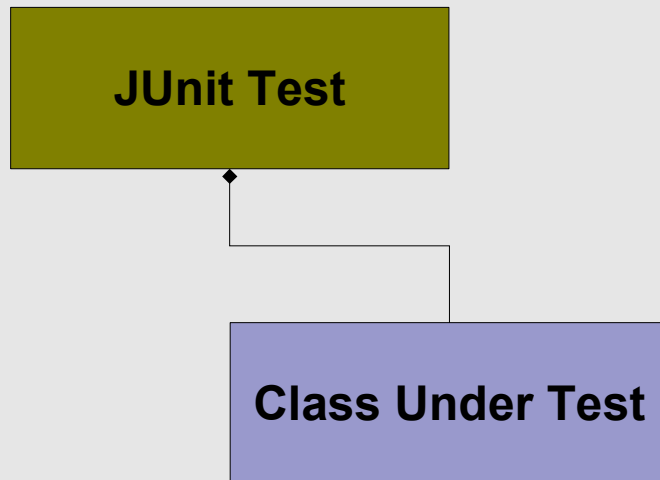
TDD and Beyond



TDD Evolution

- *No testing*
- *Test code in your main()*
- *Home grown test suites*
- *JUnit half-step*
- *JUnit*
- *Leveraging JUnit with additional tools*

- *Lightweight*
- *Effective*
- *Assertions*



The screenshot displays the JUnit test runner interface. At the top, it shows the Package Explorer, Navigator, and JUnit icons. Below this, it indicates the test run is "Finished after 1.187 seconds". A progress bar shows the test run status, with a green bar indicating success. The summary statistics are: "Runs: 8/8", "Errors: 0", and "Failures: 0". The "Failures" tab is selected, showing a tree view of the test hierarchy. The hierarchy includes:

- JUnit Test Suite
 - net.sf.javasandbox.junitperf.FactorialTest
 - testCalcRecursive
 - testCalcRecursiveInputTooLarge
 - testCalcIterative
 - testCalcBruteForce
 - net.sf.javasandbox.junit.j2se.GenericsTest
 - net.sf.javasandbox.hsqldb.UserDaoTest
 - testCreate
 - testRead
 - net.sf.javasandbox.easymock.UserControllerTest

JUnit Code Sample

- *Extend TestCase*
- *setUp()*
- *tearDown()*
- *Test methods are named test*()*
- *assert*();*

```
public void test()
{
    List<String> stringList = new Vector<String>();
    String string1 = "hello";
    String string2 = "world";

    stringList.add(string1);
    stringList.add(string2);

    // we can access the list without casting
    assertEquals(string1, stringList.get(0));
    assertEquals(string2, stringList.get(1));

    assertTrue(stringList.contains("hello"));

    // the vector is forgiving if you use a different type
    assertEquals(-1, stringList.lastIndexOf(new Long(1)));

    assertEquals("[hello, world]", stringList.toString());
}
```

Unit Tests in Construction

- *Write a test to the interface you'd like to use*
- *Implement the interface and code until the tests pass*
- *Keep repeating for additional functionality*

Unit Tests in Refactoring

Good unit tests give you confidence for refactoring

- *Make sure tests pass before beginning*
- *Refactor in small steps testing along the way*
- *Make sure all tests pass after refactoring*

Unit Tests in Maintenance

- *Change your attitude about defects*
- *Defects mean there's a test that you haven't thought of*
- *Create the test and observe it failing*
- *Fix the defect*

What Makes a good Test?

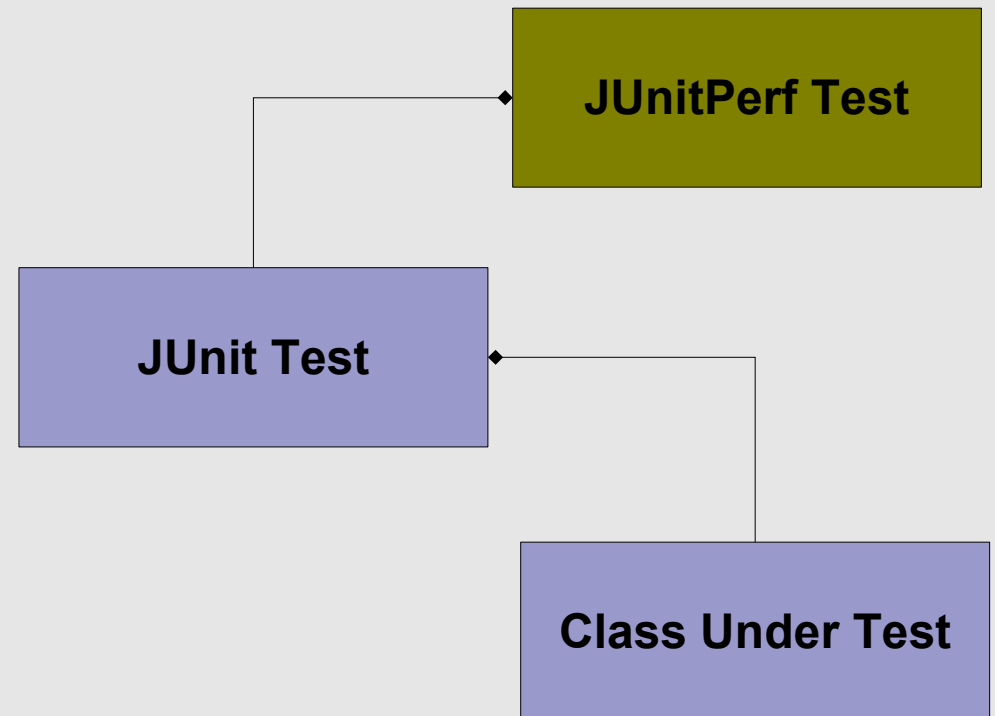
- *Isolated*
- *Repeatable*
- *Easy to run*
- *Quick*
- *Self-checking (assertions)*
- *Happy path & edge cases*

Assertions, Assertions, Assertions

- *If you aren't using assertions then you aren't using JUnit to it's full potential*
- *Humans are expensive*
- *Humans don't scale*
- *Humans are forgetful*
- *If you (the human) are looking at logs to determine if your tests passed you need to add more assertions!*

JUnitPerf

- *Can test for performance and load*
- *Write JUnits as you normally would*
- *Wrap your JUnit tests with JUnitPerf*



JUnitPerf Code Sample

```
private static long MAX_ELAPSED_TIME = 100;

public static Test suite() {
    TestSuite suite = new TestSuite();

    // check performance of recursive method
    Test factorialRecursiveTest = new FactorialTest("testCalcRecursive");
    suite.addTest(new TimedTest(factorialRecursiveTest, MAX_ELAPSED_TIME));

    // check performance of iterative method
    Test factorialIterativeTest = new FactorialTest("testCalcIterative");
    suite.addTest(new TimedTest(factorialIterativeTest, MAX_ELAPSED_TIME));

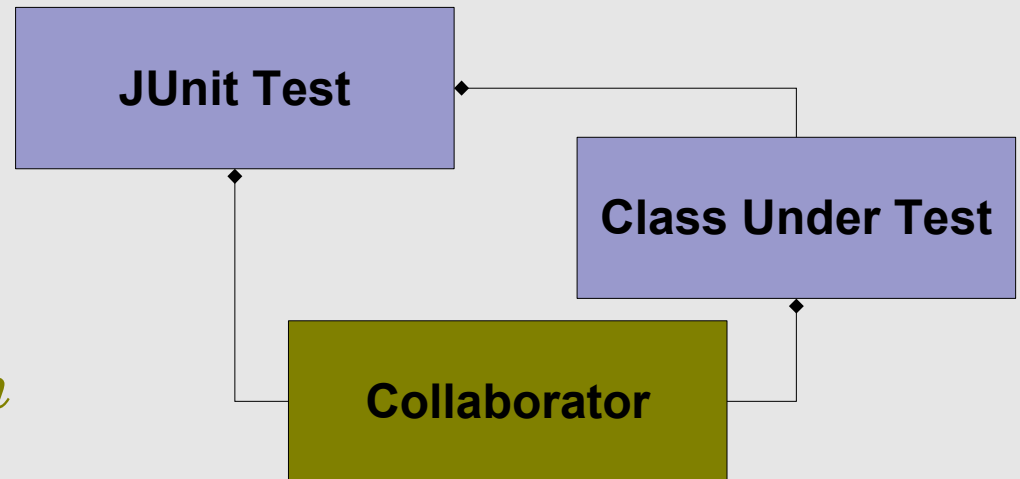
    // check performance of brute force method
    Test factorialBruteForceTest = new FactorialTest("testBruteForce");
    suite.addTest(new TimedTest(factorialBruteForceTest, MAX_ELAPSED_TIME));

    return suite;
}
```

EasyMock

- *Types of mocking*

- *hand coded*
- *build-time generation*
- *dynamic (EasyMock)*



- *You want to test something but the class under test has a collaborator.*
- *The collaborator is what you will mock!*
- *Encourages dependency injection*

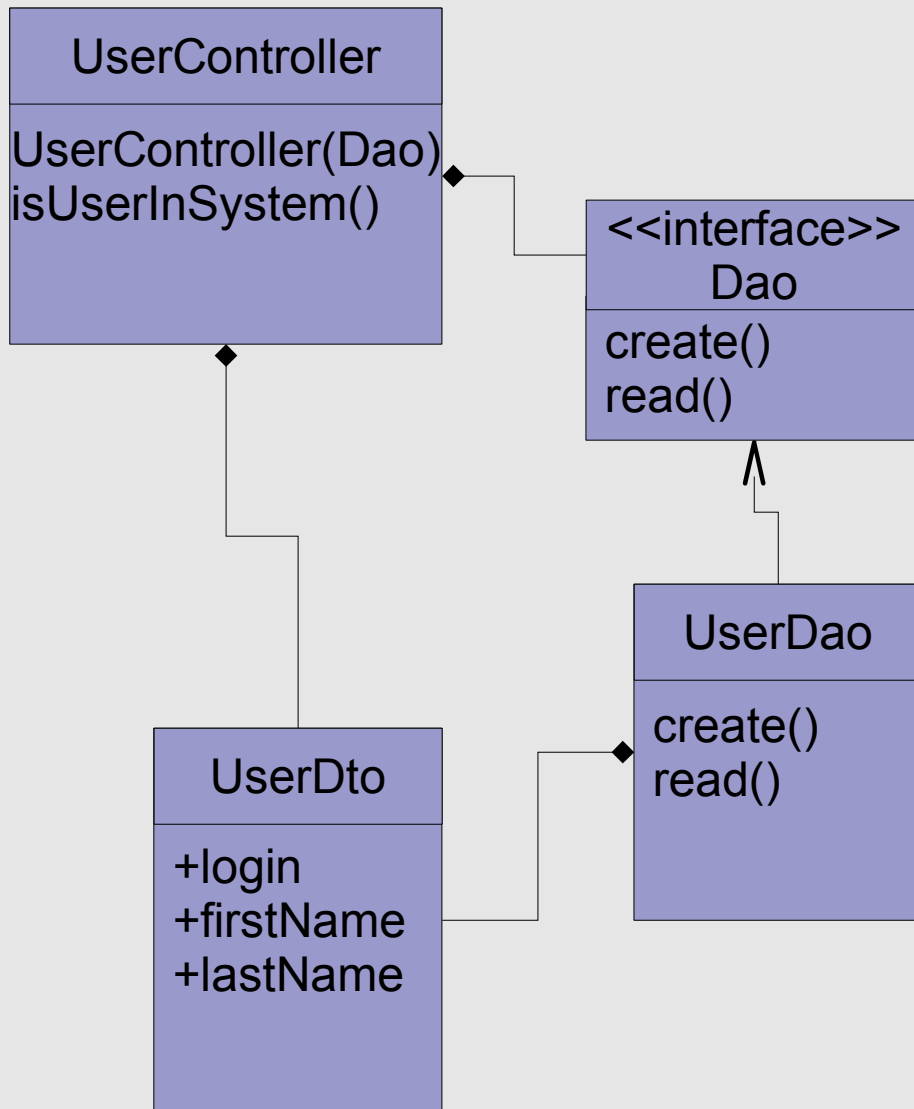
When to Mock

- *Dealing with collaborators makes it cumbersome to write tests*
- *'Out of container' testing*
- *You want to eliminate the need to have a database, network or other equipment available*
- *You want to reliably produce behavior that is difficult to create without a mock; too many rows exception, etc.*

Downside of Mocks

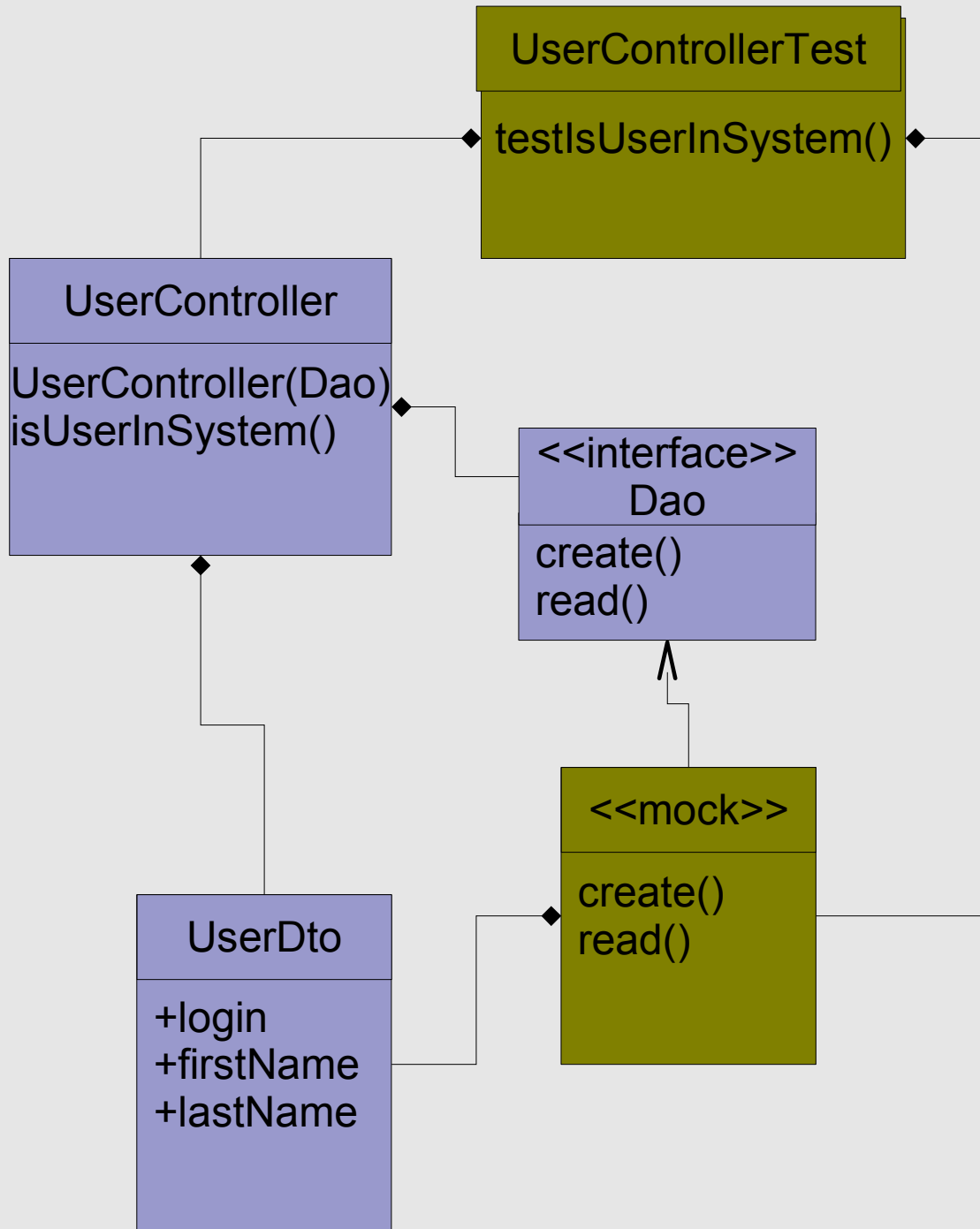
- *Test code becomes ugly*
- *The APIs aren't intuitive (to me)*
- *Most frameworks only allow you to mock Java interfaces*
- *Sometimes confusing to figure out what to mock (for me)*

System as Designed



- *We're doing an MVC UI*
- *UserController talks to a persistence framework*
- *We want to test our controller*
- *Don't want to test DAO, database, network*

W/ Mocks



- *JUnit test creates a mock and gives it to UserController*
- *The DAO implementation is replaced with a mock*

EasyMock Code Sample

```
import static org.easymock.EasyMock.createMock;

public class UserControllerTest extends TestCase
{
    private List<UserDto> getUserDtoList()
    {
        List<UserDto> listOut = new Vector<UserDto>();

        UserDto currUserDto = new UserDto();
        currUserDto.setLogin("dolbie");
        currUserDto.setFirstName("Dolbie");
        currUserDto.setLastName("Smith");
        listOut.add(currUserDto);

        return listOut;
    }

    public void testIsUserInSystem() throws Exception
    {
        Dao mockUserDao = createMock(Dao.class);
        UserController userController = new UserController(mockUserDao);
        expect(mockUserDao.read()).andReturn(getUserDtoList());
        replay(mockUserDao);

        String loginToCheck = "dolbie";
        boolean result = userController.isUserInSystem(loginToCheck);

        assertTrue("couldn't find " + loginToCheck, result);
    }
}
```

