CSCI 2120:
Software Design & Development II

Testing, Unit Testing & JUnit

Testing

Importance of Testing

- Testing is a ubiquitous and expensive software engineering activity
  - It is not unusual to spend 30-40% of total project effort on testing
  - For big and/or life-critical systems (e.g., flight control), testing cost can be several times the cost of all other software engineering activities combined

Different Types of Testing

- TWO MAIN TYPES OF TESTING:
- **Functional Testing** – Given some input, does the software system *as a whole* produce correct output? Often referred to as "black box" testing
- **Unit Testing** – Do each of the *individual components* of your software system behave properly?

Black vs. White vs. Grey Box Testing

- White Box Testing (Implementation-Driven Testing
  - We have full access to the code we are testing.
  - We can use the code to get an idea of how to test.
  - Often development-driven by the authors of the code being tested.
- Black Box Testing (Functional Testing)
  - We only have the documentation for the source.
  - We test the code using what we expect to happen based on the classes' contracts.
- Grey Box Testing (Test-Driven Implementation)
  - We “have” the source, but we haven’t implemented it yet.
  - We write the code to pass the test which it must pass.

Unit Testing: Dealing with Scale

- **Best practice** is to test individual ***units*** or ***components*** of software (one class, one method at a time)
  - This is known as ***unit testing***
  - Testing what happens when multiple components are put together into a larger system is known as ***integration testing***
  - Testing a whole end-user system is known as ***system testing***

Motivations of Testing in Individual Environments

- After designing a solution to a problem and implementing, the role of testing is to verify that your approach to a solution works as expected
- Testing is partly what makes computer science a "science"
- Without verifying your results, it can lead to unforeseen erroneous behavior.

Motivation of Testing in Team-based Environments

- An innate problem in any team-based coding endeavor is in the distributed nature.

- Typically a team consist of a software architect who communicates with a team of lead developers who then communicate with pods of junior developers. The workload of the coding must be divided up to each person.

- How can you ensure that everyone is on the same page? That everyone understands the complexity of the system? You can't. Such an approach is doomed to failure.

Role of Testing in Team-based Environments

- The solution to this distributed problem is in testing. Most of the team only needs a small understanding of the system. Each member will be a implementer to part of the system and the clients to the rest via the design by contract approach.

- Thus the system is broken into several small class/method APIs. To ensure that each dev understands their contract (preconditions/postconditions) the first task is to build test-cases. Given this input what should the valid output be & approves these with the project lead. Then they develop their code to pass those test. If its passes the test it should seamlessly integrate into the software system.

- This helps assign responsibility across the dev team. Because if the submitted code does break the system but it passed all the approved testing, then the fault is with the manager for approving the testset as it is supposed to mock the actual usage of that class/method in a deployed state.

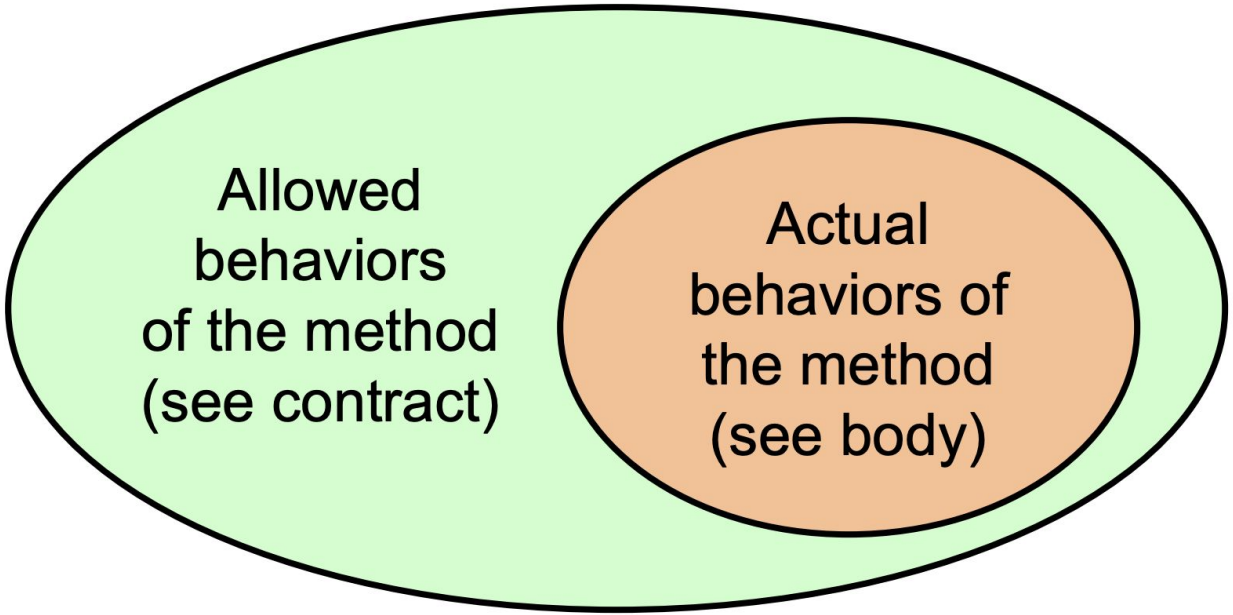
Testing Functional Correctness

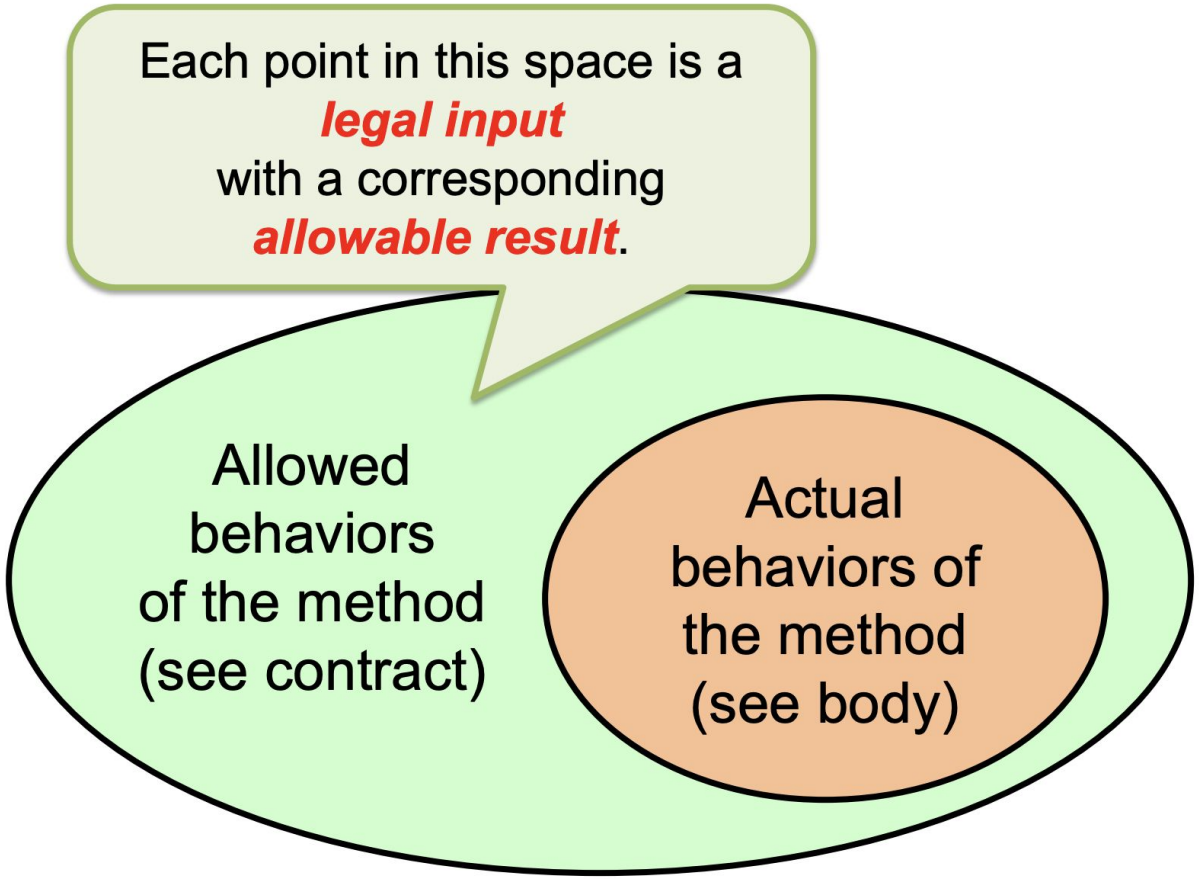
- What does it mean for a program unit (let's say a method) to be ***correct***?
  - It does what it is supposed to do.
  - It doesn't do what it is not supposed to do.

“Supposed To Do”?

- How do we know what a method is supposed to do, and what it is not supposed to do?
  - We look at its ***contract***, which is a ***specification*** of its ***intended behavior***

Behaviors


Allowed
behaviors
of the method
(see contract)
Actual
behaviors of
the method
(see body)


Each point in this space is a
legal input
with a corresponding
allowable result.
Allowed
behaviors
of the method
(see contract)
Actual
behaviors of
the method
(see body)

Example Method Contract

```
/**
 * Reports some factor of a number.
 * ...
 * @requires
 * n > 0
 * @ensures
 * aFactor > 0  and
 * n mod aFactor = 0
*/
private static int aFactor(int n) {...}
```

Example Method Contract

```
/**
 * Reports some facto
 * ...
 * @requires
 * n > 0
 * @ensures
 * aFactor > 0  and
 * n mod aFactor = 0
*/
private static int aFactor(int n) {...}
```

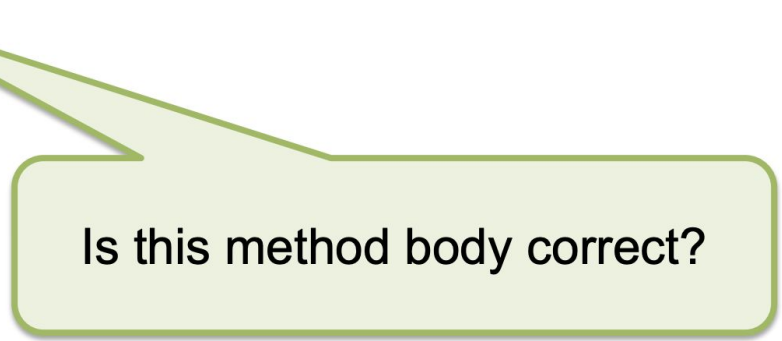
This means:
"*n* is divisible by *aFactor*".

Example Method Body

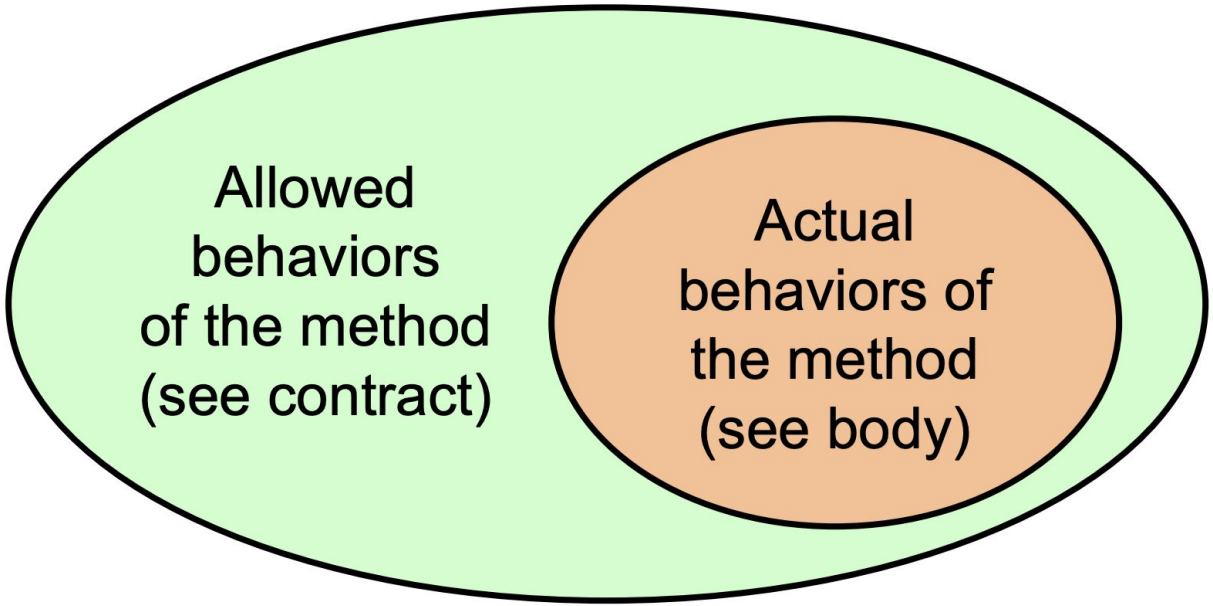
```
private static int aFactor(int n) {
  return 1;
}
```

Example Method Body

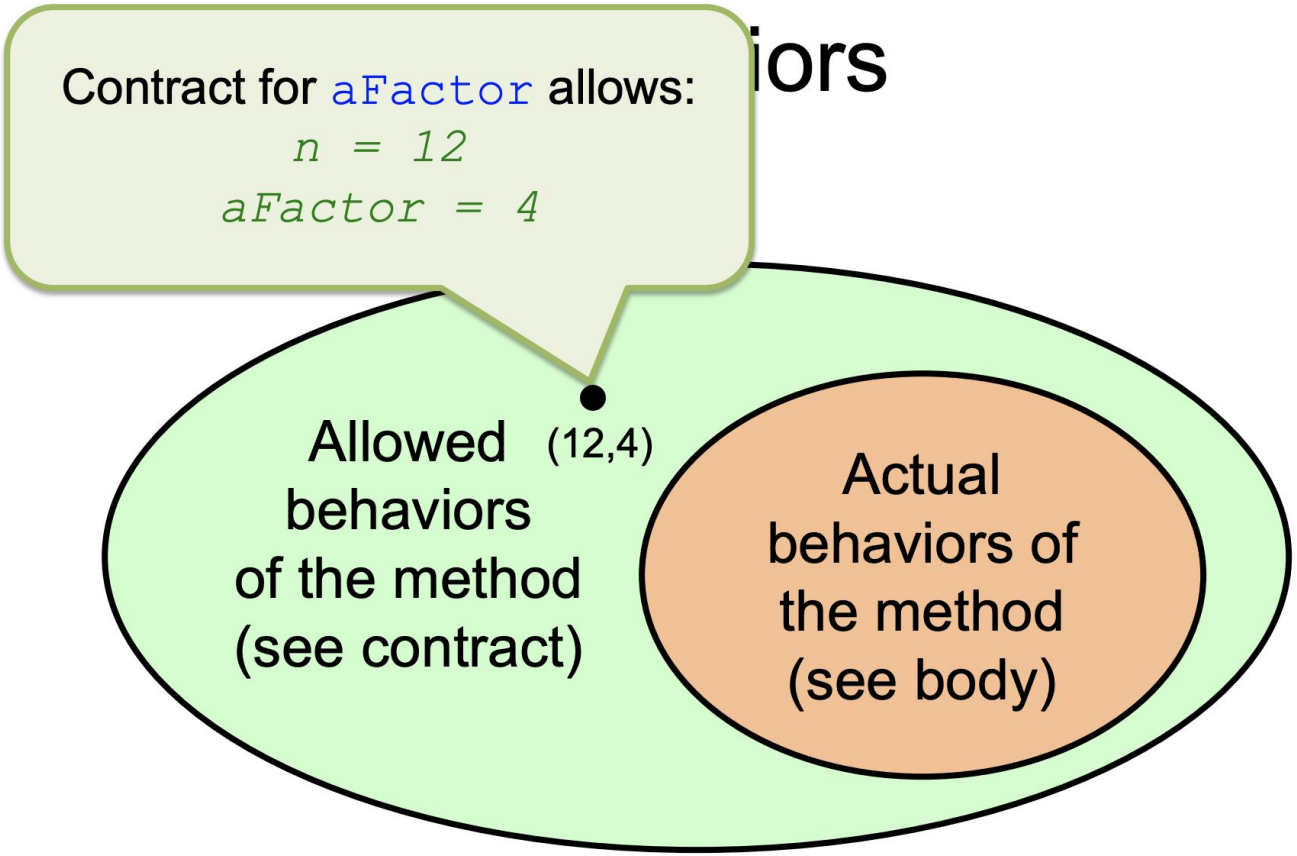
```
private static int aFactor(int n) {
   return 1;
}
```

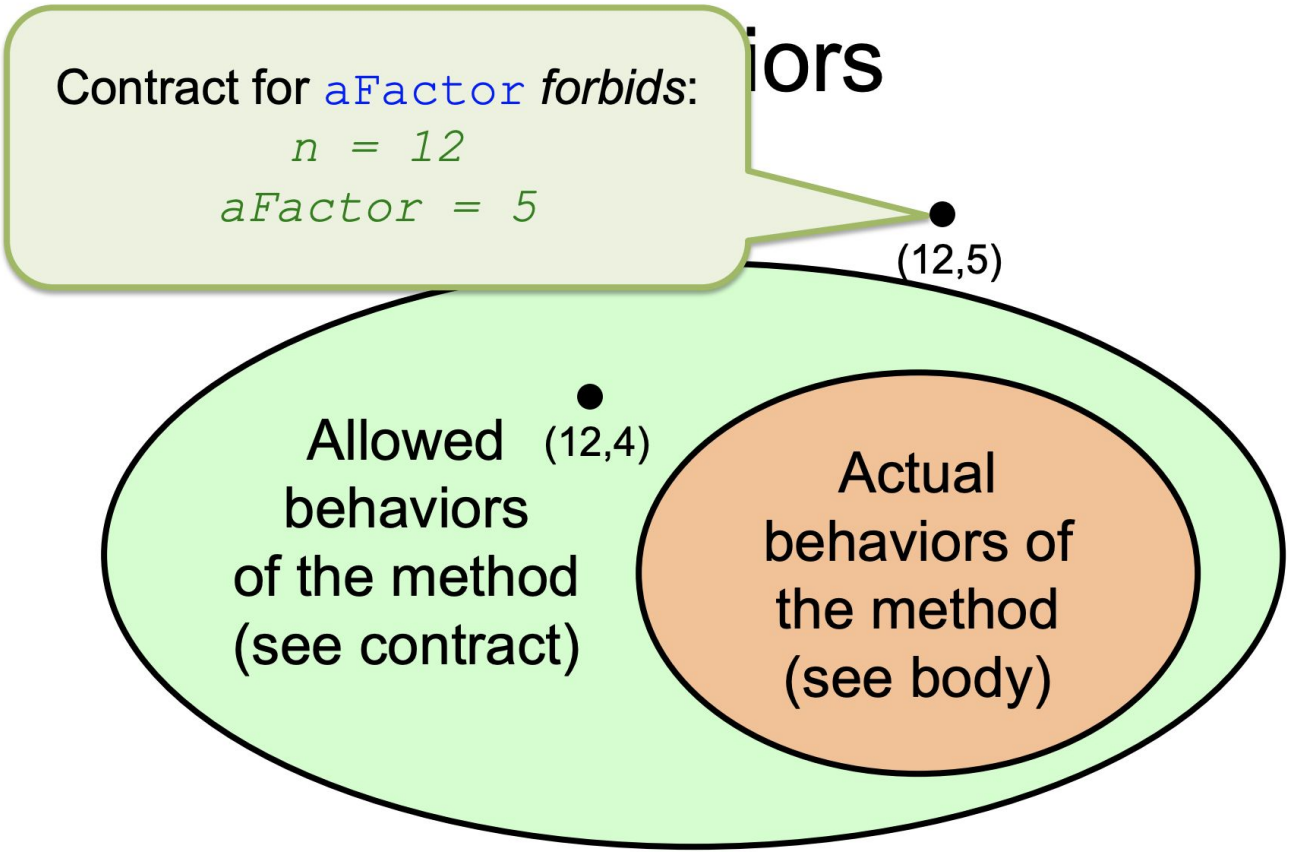


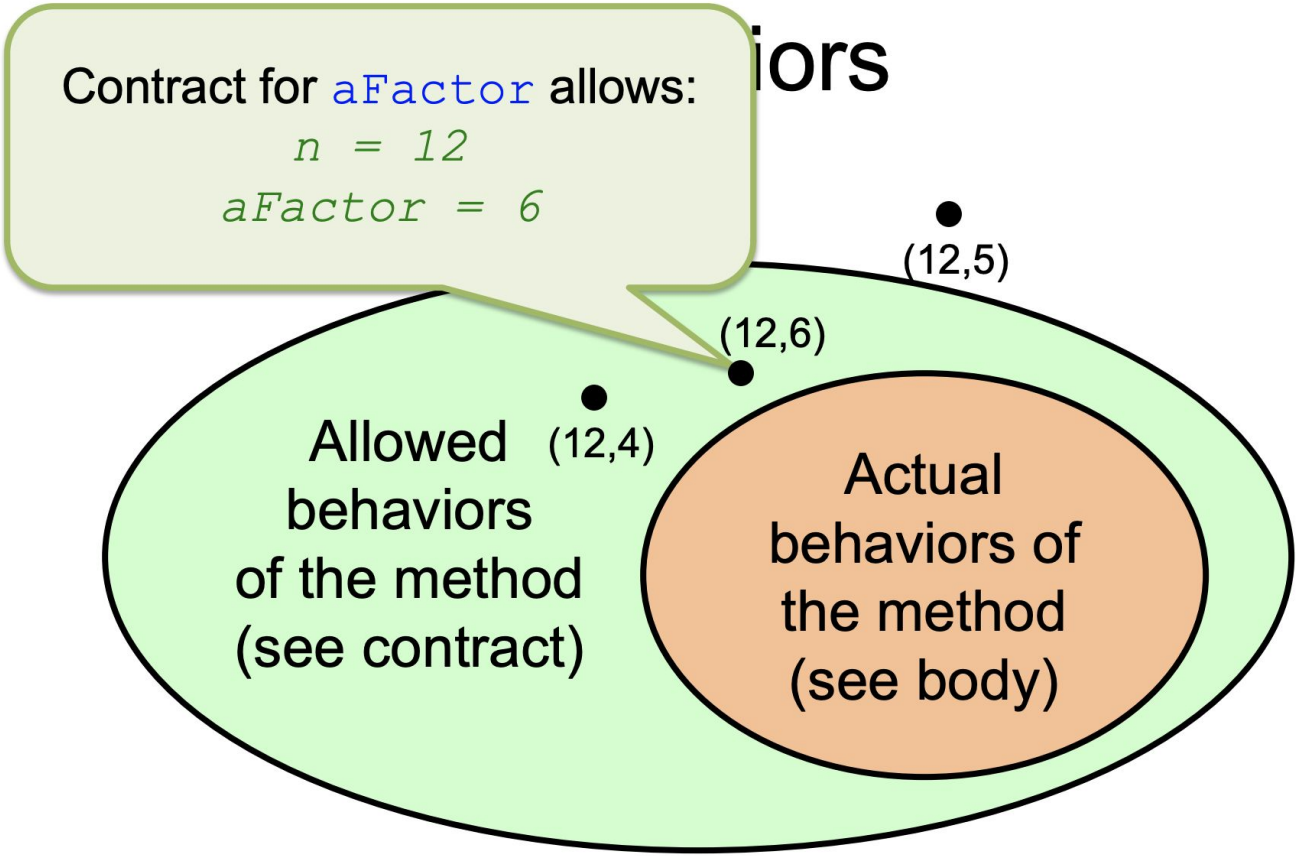
Behaviors

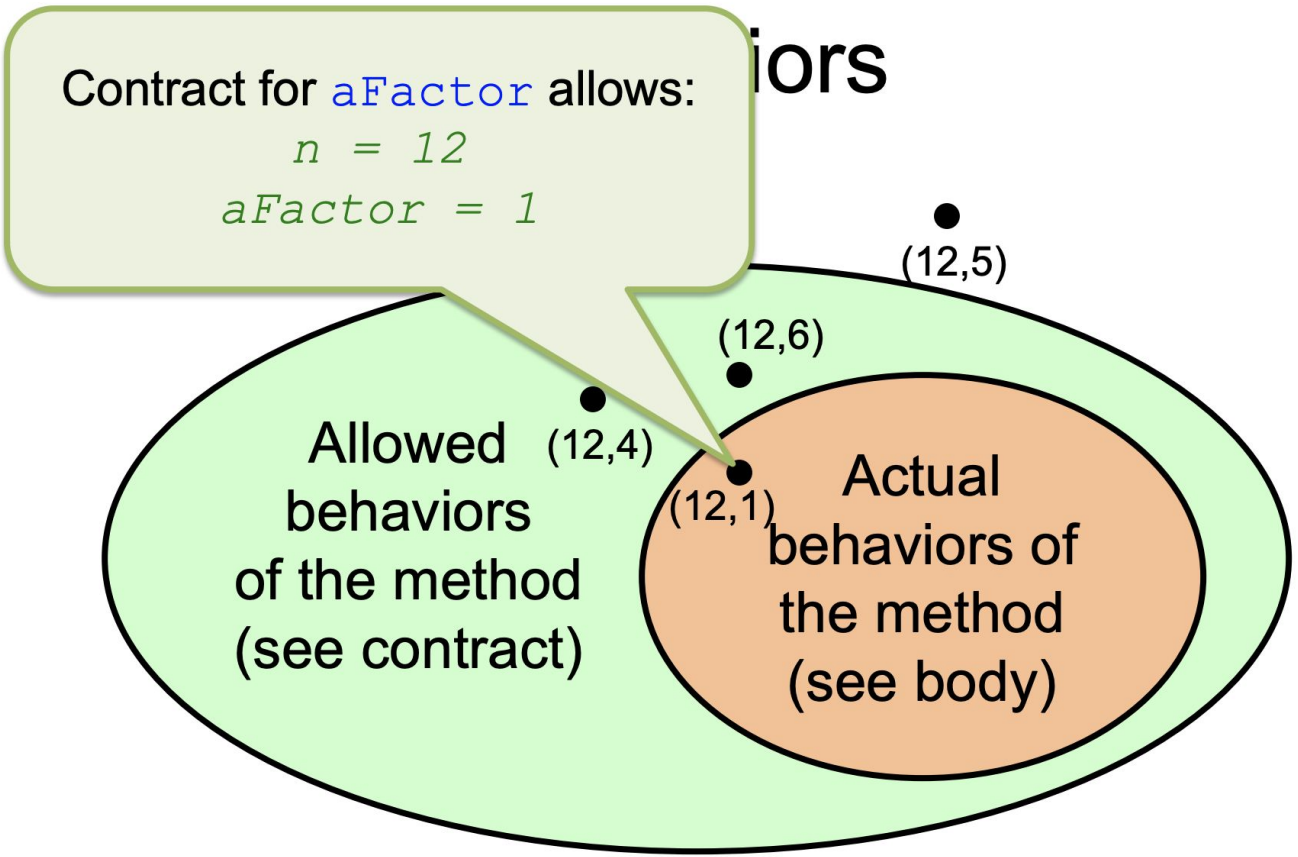

Allowed
behaviors
of the method
(see contract)
Actual
behaviors of
the method
(see body)

iors

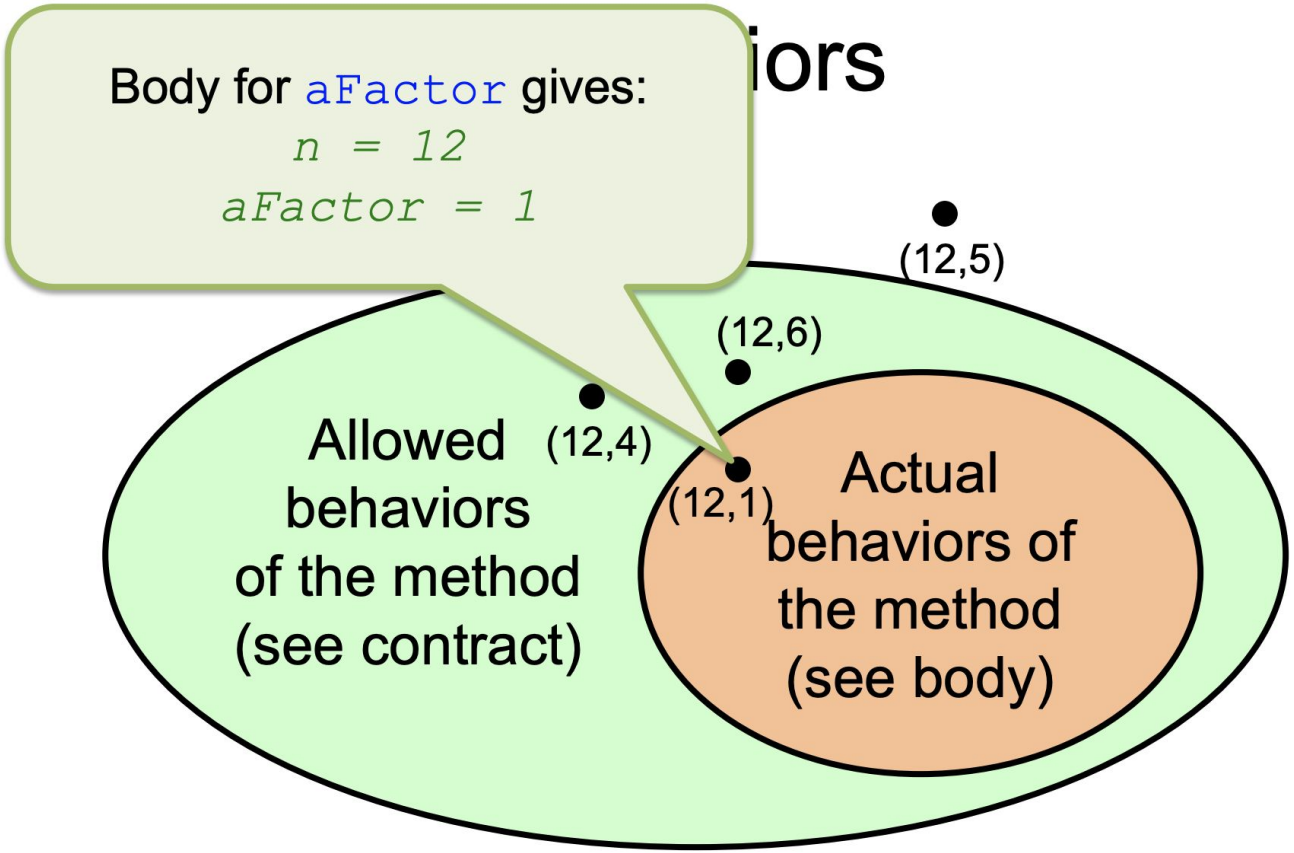

Contract for aFactor allows:
n = 12
aFactor = 4
Allowed behaviors of the method (see contract)
(12,4)
Actual behaviors of the method (see body)


iors
Contract for aFactor forbids:
n = 12
aFactor = 5
(12,5)
(12,4)
Allowed behaviors of the method (see contract)
Actual behaviors of the method (see body)


Contract for aFactor allows:
n = 12
aFactor = 6
(12,5)
(12,6)
(12,4)
Allowed behaviors of the method (see contract)
Actual behaviors of the method (see body)

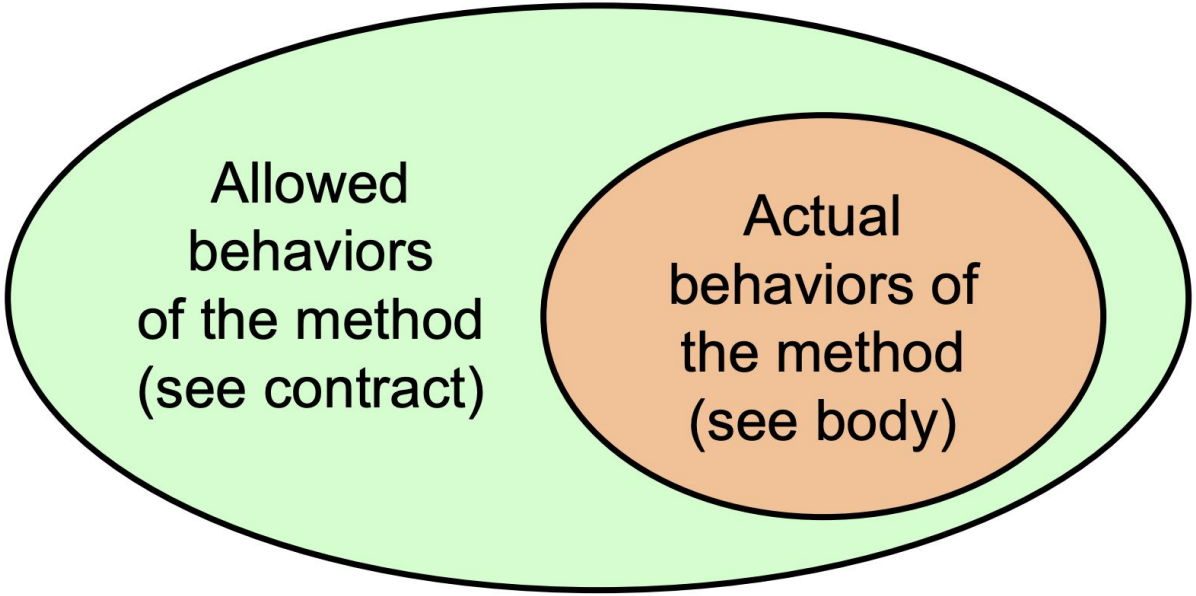

Contract for aFactor allows:
n = 12
aFactor = 1
(12,5)
(12,6)
(12,4)
(12,1)
Allowed behaviors of the method (see contract)
Actual behaviors of the method (see body)

iors


Body for aFactor gives:
n = 12
aFactor = 1
(12,5)
(12,6)
(12,4)
(12,1)
Allowed behaviors of the method (see contract)
Actual behaviors of the method (see body)

Definition of Correctness

- Body is ***correct*** if *actual* is a subset of *allowed*.



“Implements” Revisited

- If you write `class C implements I`, the Java compiler checks that for each method in `I` there *is* some method body for it in `C`
- We really care about much more: that for each method in `I` the method body for it in `C` is ***correct*** in the sense just defined

“Implements” Revisited

- If you write **`cla`** the Java compi… method in `I` there … method body for it in `C`
- We really car… about much more: that for each meth…d in `I` the method body for it in `C` is ***correct*** in the sense just defined

How can you decide whether this is the case for a given method body?

Testing

- ***Testing*** is a technique for trying to ***refute the claim*** that a method body is correct for the method contract
- In other words, the ***goal*** of testing is to show that the method body does *not* correctly implement the contract, i.e., that it is *defective*
  - As a tester, you really want to think this way!

Psychology of Testing

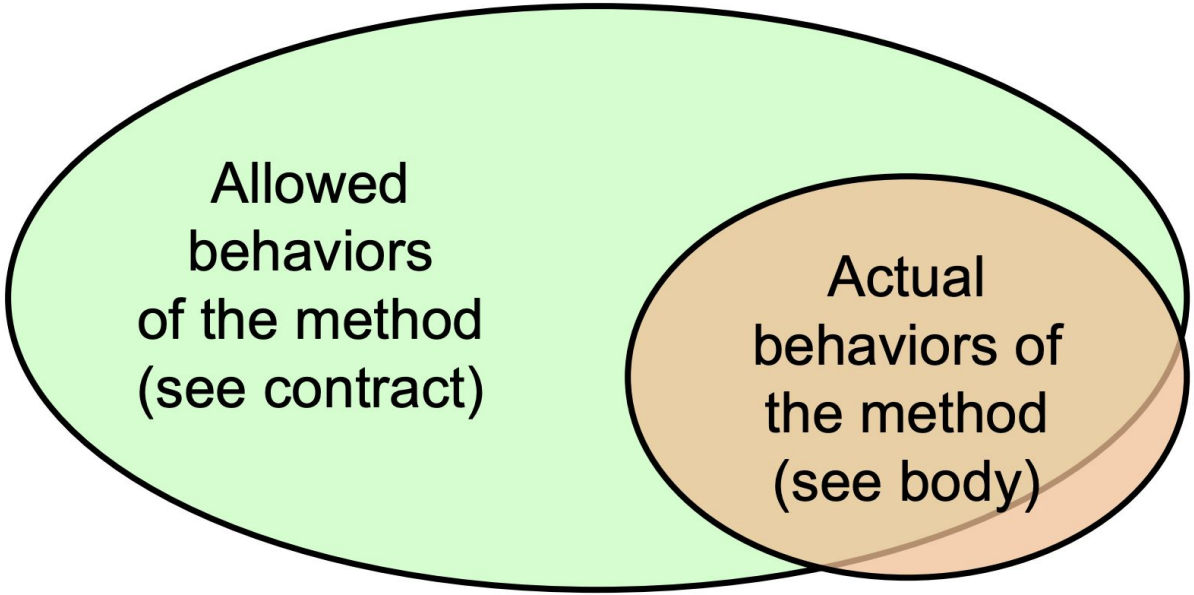
- Design and coding are ***creative*** activities
- Testing is a ***destructive*** activity
  - The primary goal is to "break" the software, i.e., to show that it has defects
- Very often the same person does both coding and testing (*not* a **best practice**)
  - You need a "split personality": when you start testing, become paranoid and malicious
  - It's surprisingly hard to do: people don't like finding out that they made mistakes

Testing vs. Debugging

- Goal of ***testing***: given some code, show by executing it that it has a defect (i.e., there is at least one situation where the code's actual behavior is not an allowed behavior)
- Goal of ***debugging***: given some source code that has a defect, find the defect and repair it

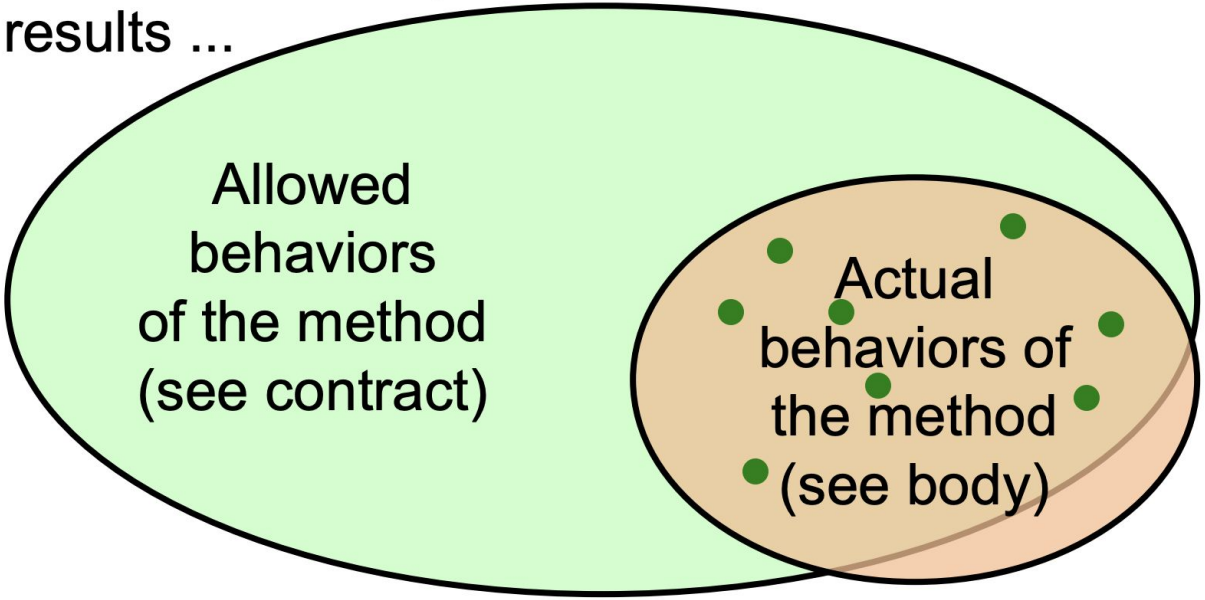
Incorrect (Defective) Code

- If actual behaviors *are not* a subset of allowed...



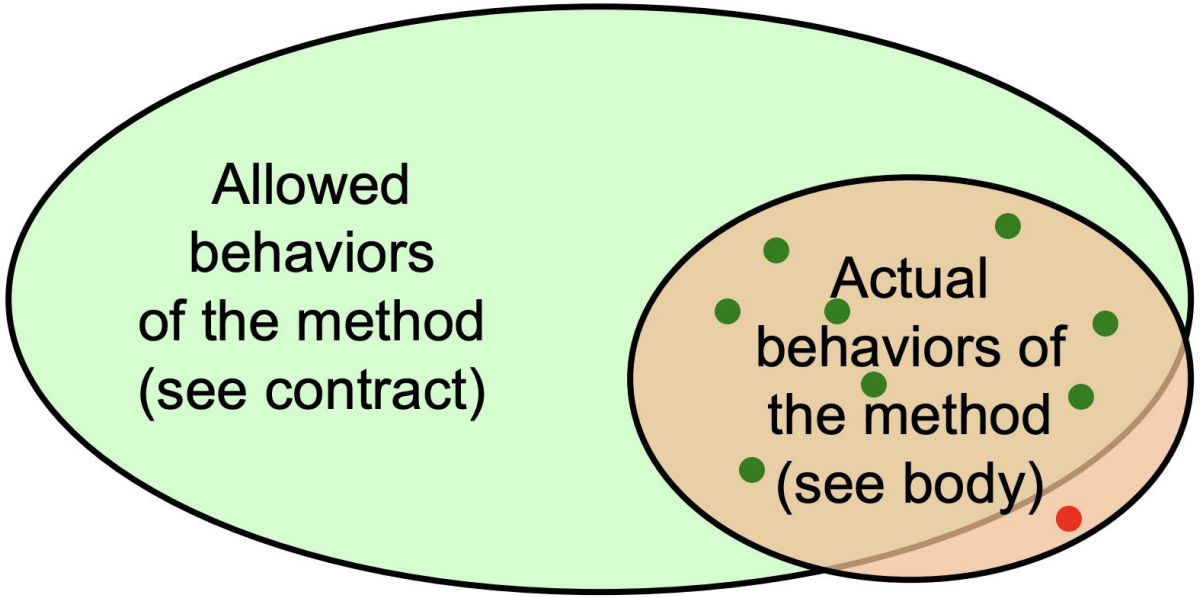
Incorrect (Defective) Code

- ... and we start trying some inputs and observing results ...



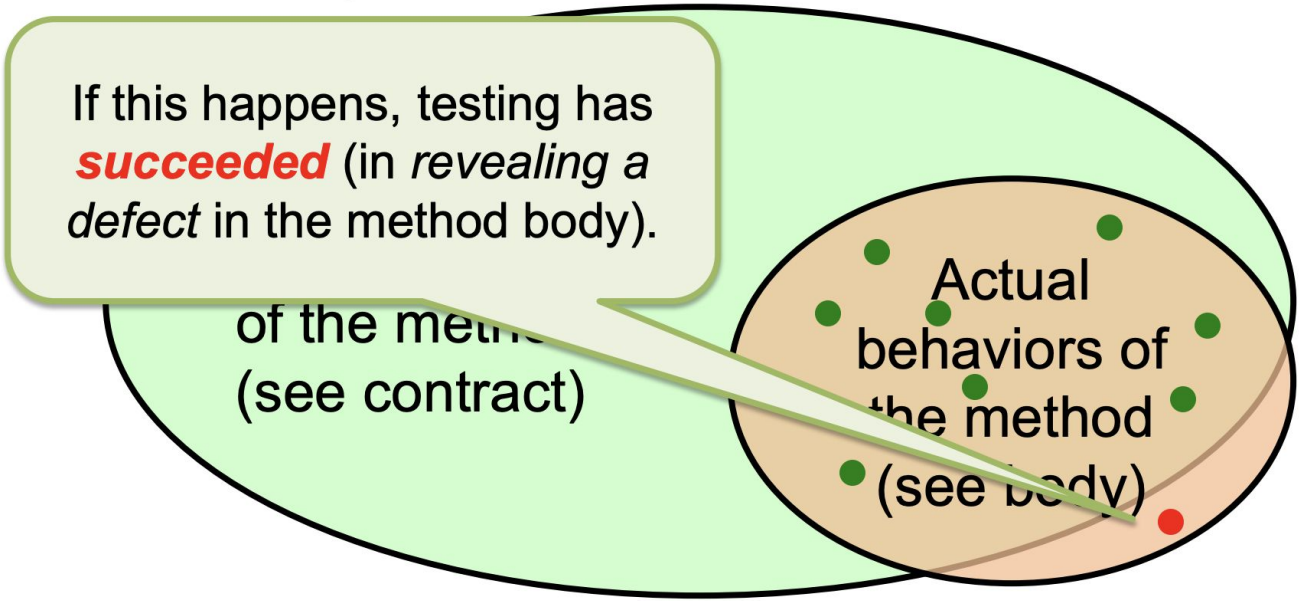
Incorrect (Defective) Code

- ... one might lie outside the allowed behaviors!



Incorrect (Defective) Code

- ... one might lie outside the allowed behaviors!



Test Cases

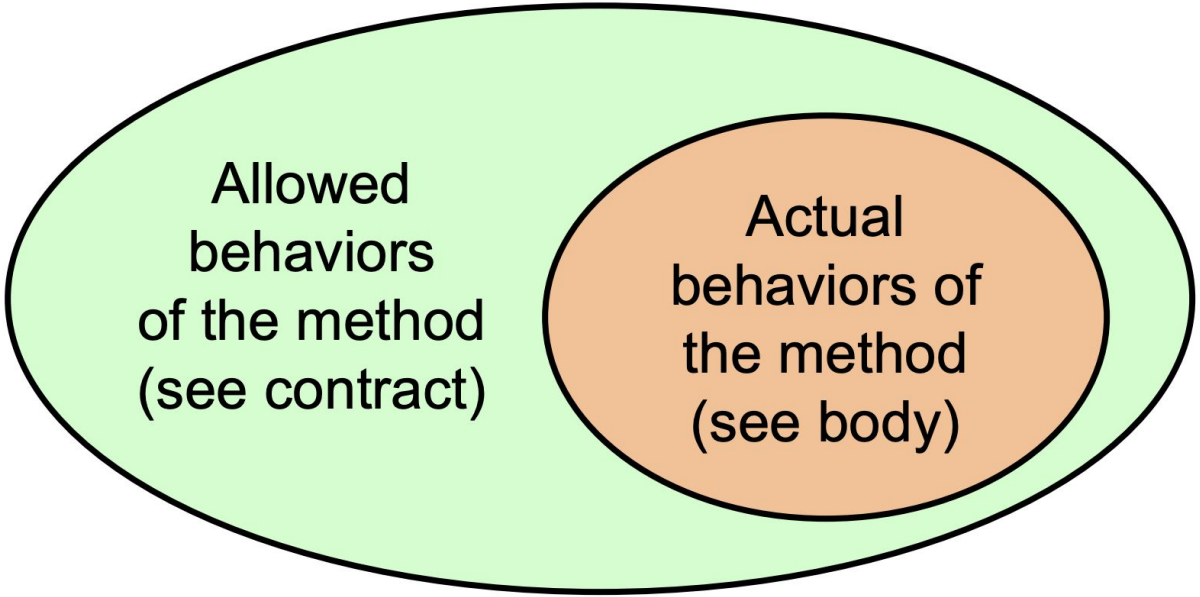
- Each input value and corresponding allowed/expected result is a ***test case***
- Test cases that do *not* reveal a defect in the code do not help us refute a claim of correctness
- Test cases like that last one should be cherished!

Test Plan/Test Fixture

- A set of test cases for a given unit is called a ***test plan*** or a ***test fixture*** for that unit

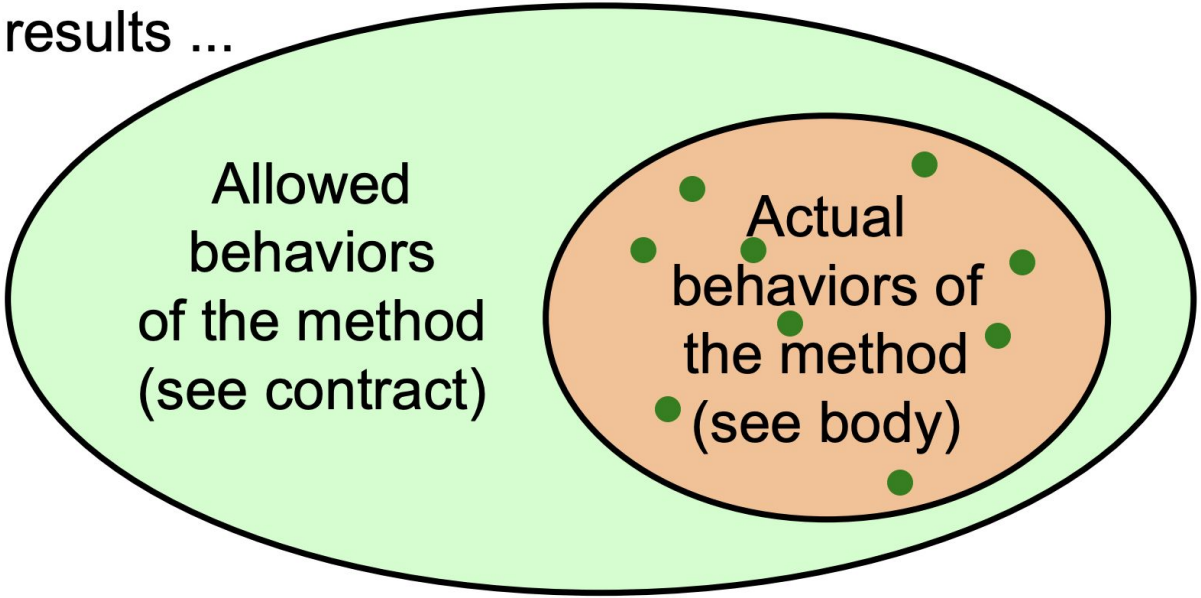
Correct Code

- If actual behaviors are a subset of allowed...



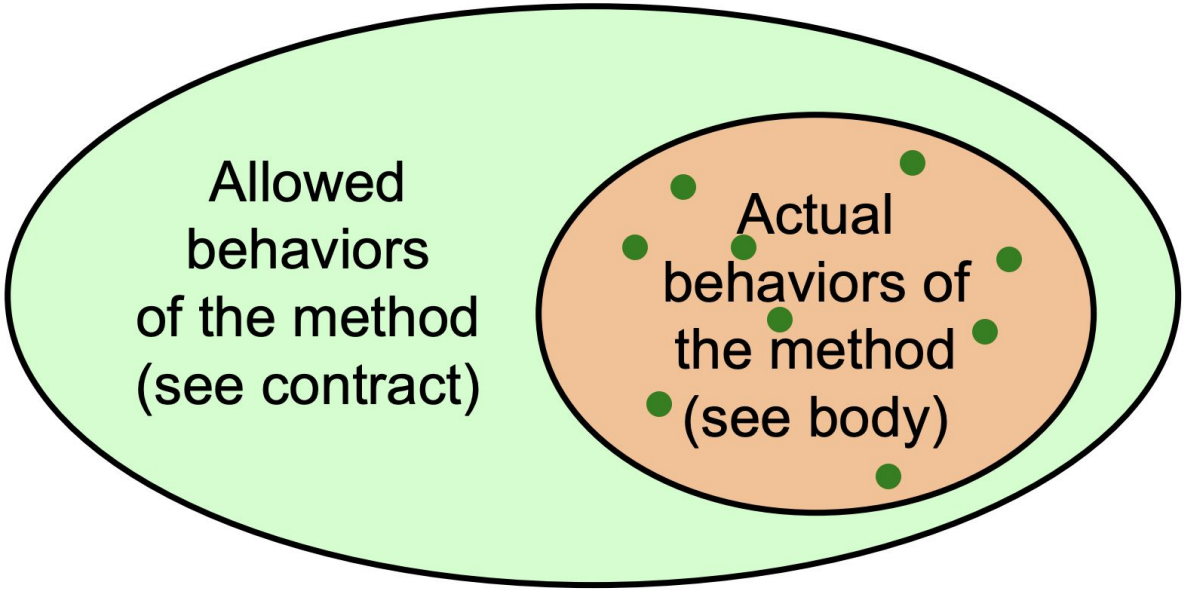
Correct Code

- ... and we start trying some inputs and observing results ...



Correct Code

- ... then we will never find a defect.



Severe Limitation of Testing

- “Program testing can be used to show the presence of bugs, but never to show their absence!”

— *Edsger W. Dijkstra (1972)*

Designing a Test Plan

- To make testing most likely to succeed in revealing defects, **best practices** include:
  - Test ***boundary*** cases: “smallest”, “largest”, “special” values based on the contract
  - Test ***routine*** cases
  - Test ***challenging*** cases, i.e., ones that, if *you* were writing the code (maybe you didn’t write the code being tested!), *you* might find difficult or error-prone

Example Method Contract #1

```
/**
 * Returns some factor of a number.
 * ...
 * @requires
 * n > 0
 * @ensures
 * aFactor > 0  and
 * n mod aFactor = 0
 */
private static int aFactor(int n) {...}
```

Partial Test Plan

Inputs	Results	Reason
`n = 1`	`aFactor = 1`	boundary
`n = 2`	`aFactor = 1` `aFactor = 2`	routine challenging? (prime)

`n = 4`	`aFactor = 1`
	`aFactor = 2`
	`aFactor = 4`
		challenging? (square)

`n = 12`	`aFactor = 1`
	`aFactor = 2`
	`aFactor = 3`
	`aFactor = 4`
	`aFactor = 6`
	`aFactor = 12`
		routine

Example Method Contract #2

```
/**
 * Decrements the given NaturalNumber.
 * ...
 * @updates n
 * @requires
 * n > 0
 * @ensures
 * n = #n - 1
 */
private static void decrement(NaturalNumber n) {...}
```

Partial Test Plan

Inputs	Results	Reason
`#n = 1`	`n = 0`	boundary
`#n = 2`	`n = 1`	routine
`#n = 10`	`n = 9`	challenging? (borrow)
`#n = 42`	`n = 41`	routine

Partial Test Plan

Inputs	Results	Reason
`#n = 1`	`n = 0`	boundary
`#n = 2`	`n = 1`	routine
`#n = 10`	`n = 9`	challenging? (borrow)
`#n = 42`	`n = 41`	routine
`#n = 0`		

What about this “boundary” case, which is on the illegal side of the “boundary” between legal and illegal inputs?

Partial Test Plan

Inputs	Results	Reason
`#n = 1`	`n = 0`	boundary
`#n = 2`	`n = 1`	routine
`#n = 10`	`n = 9`	challenging? (borrow)
`#n = 42`	`n = 41`	routine
`#n = 0`		

This test case is worthless: it violates the requires clause, so it *cannot possibly reveal a defect* in the method body. Why not?

JUnit

Primitive Testing

- Write `main` as a ***command interpreter*** with console input/output, so user (tester) provides inputs and observes actual results (as in some recent lab skeletons)
- Tester compares actual results with allowed/expected results by ***inspection***
- Pros/cons:
  - Simple, easy, intuitive
  - Tedious, error-prone, not automated

Example

```
String command = getCommand(in, out);
while (!command.equals("q")) {
  if (command.equals("i")) {
    out.print("Enter a natural number: ");
    NaturalNumber n =
        new NaturalNumber2(in.nextLine());
    out.println("Before increment: n = " + n);
    increment(n);
    out.println("After increment:  n = " + n);
  } else if (command.equals("d")) {...}
  command = getCommand(in, out);
}
```

More Automated Testing

- Write `main` to contain sets of inputs and expected results in “parallel arrays” of argument values and expected results (as in some other recent lab skeletons)
- Simple loop in `main` compares actual results with allowed/expected results
- Pros/cons:
  - Better, primarily because the process is now far more automatic

Example

```
final int[] numbers = { 0, 0, 1, 82, 3, 9, 27, 81, 243 };
final int[] roots = { 1, 2, 3, 2, 17, 2, 3, 4, 5};
final int[] results = { 0, 0, 1, 9, 1, 3, 3, 3, 3 };
for (int i = 0; i < numbers.length; i++) {
  int x = root(numbers[i], roots[i]);
  if (x == results[i]) {
    out.println("Test passed: root(" + numbers[i]
        + ", " + roots[i] + ") = " + x);
  } else {
    out.println("*** Test failed: root(" + numbers[i]
        + ", " + roots[i] + ") expected " + results[i]
        + " but was " + x);
  }
}
```

Remaining Problems

- One new drawback of this approach is that you need to be able to write the values of the arguments and expected results using Java literals in the array initializations
  - This does not work for some types, where each set of input values and/or expected results must be created by performing a series of method calls

Remaining Problems

- Another drawback of this approach is that, if there are multiple allowed results for the given arguments, mere equality checking with actual results *does not work*
  - Recall the `aFactor` method; what happens if we write in the `results` array that *the* "expected" result is 6, when any of 1, 2, 3, or 6 (and maybe other results) are *also* allowed?

Serious Testing: JUnit

- ***JUnit*** is an industry-standard “framework” for testing Java code
  - A ***framework*** is one or more components with “holes” in them, i.e., some missing code
  - Programmer writes classes following particular conventions to fill in the missing code
  - Result of combining the framework code with the programmer’s code is a complete product

Automating Tests

- Typically, each class will have one JUnit test class associated with it.
  - i.e., class Animal is tested by class AnimalTest.
- This test class can then be run at any time.
- The idea is to write tests that ensure the class is working the way it ought to.
  - Do normal cases work?
  - Do edge cases work?
  - Are errors handled correctly?

Methods

- In Java, the “units” of a class being tested are usually its *methods*.
- Since methods should be small and serve only a single purpose, they lend themselves well to testing.
- A good test class will have multiple tests per method that properly measure if the method is behaving.

Testing

- It is a good idea to write your tests such that specific input values are provided and compared against specific expected output values.
- If the values agree, we can assume that the test is successful.
- A good test will test each method using multiple test cases specifically chosen to benchmark edge cases.

Advantages of Testing

- No more testing “by hand.”
- Can easily check if a new feature breaks the tests.
  - Allow large new changes to rapidly be made.
  - Check if change requests from other users break any part of the system.
- Can be used as a measurement of code stability.
- Help document and explain features of the code.

Example

```
import static org.junit.Assert.*;
import org.junit.Test;

public class NaturalNumberRootTest {

  @Test
  public void test1327Root3() {
    ...
  }

  ...

}
```

Example

```
import static org.junit.Assert.*;
import org.junit.Test;

public class Natura    berRootTest {

  @Test
  public voi
    ...
  }

  ...

}
```

These imports let you use JUnit features. The use of a ***static import*** allows you to call the static methods of `org.junit.Assert` without qualifying their names (see, e.g., `assertEquals` in upcoming code).

Example

```
import static org.junit.Assert.*;
import org.junit.Test;

public class NaturalNumberRootTest {

  @Test
  public void test1327Root3()
    ...
  }

  ...

}
```

A test plan or ***test fixture*** is a **`public class`** with one method per test case.

Example

```
@Test
public void test1327Root3() {
  NaturalNumber n = new NaturalNumber2(1327);
  NaturalNumber nExpected = new NaturalNumber2(1327);
  NaturalNumber r = NaturalNumber2(3);
  NaturalNumber rExpect  new NaturalNumber2(3);
  NaturalNumber rt = Natura rRoot.root(n, r);
  NaturalNumber rtExpected =  ralNumber2(10);
  assertEquals(nExpected,
  assertEquals(rExpected,
  assertEquals(rtExpected
}
```

Each ***test case*** is a **`public void`** method with no parameters.

Example

```
@Test
public void test1327Root3() {
  NaturalNumber n = new NaturalNumber2(1327);
  NaturalNumber nExpected = new NaturalNumber2(1327);
  NaturalNumber r = NaturalNumber2(3);
  NaturalNumber rExpect  new NaturalNumber2(3);
  NaturalNumber rt = Natur rRoot.root(n, r);
  NaturalNumber rtExpected =  ralNumber2(10);
  assertEquals(nExpected,
  assertEquals(rExpected,
  assertEquals(rtExpected
}
```

Each test case has an `@Test` ***annotation*** just before it.

Example

```
@Test
public void test1327Root3(
  NaturalNumber n = new Na
  NaturalNumber nExpected =
  NaturalNumber r = new Natur  mber2(3);
  NaturalNumber rExpected = new NaturalNumber2(3);
  NaturalNumber rt = NaturalNumberRoot.root(n, r);
  NaturalNumber rtExpected = new NaturalNumber2(10);
  assertEquals(nExpected, n);
  assertEquals(rExpected, r);
  assertEquals(rtExpected, rt);
}
```

There is an easy way to make a new test case: copy/paste another and then edit slightly.

Vocabulary Review

- ***Test case***
  - Exercises a single unit of code, normally a method (and a test case normally makes one call to that method)
  - Test cases should be *small* (i.e., should test one thing)
  - Test cases should be *independent* of each other
  - In JUnit: a public method that is annotated with `@Test`
- ***Test fixture***
  - Exercises a single class
  - Is a collection of *test cases*
  - In JUnit: a class that contains `@Test` methods
- Note: In Eclipse, select "New > JUnit Test *Case*" to create a new JUnit test *fixture*!

New Vocabulary

- ***(JUnit) Assertion***
  - A claim that some boolean-valued expression is true; normally, a comparison between expected and actual results (i.e., the `equals` method says they are equal)
- ***Passing a test case***
  - All JUnit assertions in the test case are *true* when the test case is executed (and no error occurred to stop program execution)
- ***Failing a test case***
  - Some JUnit assertion in the test case is *false* when the test case is executed

Execution Model

- Separate instances (objects) are created from the JUnit test fixture
  - JUnit creates one instance per test case (!)
- Implication:
  - Do not rely on order of test cases
    - Test case listed first in JUnit test fixture is not guaranteed to be executed first

JUnit Assertions

- Two most useful static methods in `org.junit.Assert` to check actual results against allowed results:

```
assertEquals (expected, actual);
assertTrue(expression);
```

- There is rarely a reason to use any of the dozens of other assertion static methods in `org.junit.Assert`

Timed Tests

- What if you're worried about an infinite loop?
  - Parameterize `@Test` with a ***timeout***: number of milliseconds before the test case is terminated for running too long

    ```
    @Test(timeout=100)
    ```

  - Problem: How do you know what is long enough for a test case to run?

Best Practices

- Some ***best practices***:
  - Keep JUnit test fixtures in the same Eclipse project as the code, but in a separate source folder (for this course: regular code in “src”, test classes/fixtures in “test”)
    - Tests are then included when project is “built”
    - Helps keep test fixtures consistent with other code

Best Practices

- Name test fixtures consistently
  - Example: class `NaturalNumberRootTest` tests class `NaturalNumberRoot`
- Name test cases consistently
  - Example: method `testFoo13` tests method `foo` with input 13

Recommended Test Case Style

```
public void test1327Root3() {
  /*
   * Set up variables and call method under test
   */
  NaturalNumber n = new NaturalNumber2(1327);
  NaturalNumber nExpected = new NaturalNumber2(1327);
  NaturalNumber r = new NaturalNumber2(3);
  NaturalNumber rExpected = new NaturalNumber2(3);
  NaturalNumber rt = NaturalNumberRoot.root(n, r);
  NaturalNumber rtExpected = new NaturalNumber2(10);
  /*
   * Assert that values of variables match expectations
   */
  assertEquals(nExpected, n);
  assertEquals(rExpected, r);
  assertEquals(rtExpected, rt);
}
```

Recommended Test Case Style

```
public void testDivideBy10NonZero() {
  /*
   * Set up variables and call method under test
   */
  NaturalNumber n = new NaturalNumber2(1327);
  NaturalNumber nExpected = new NaturalNumber2(132);
  int k = n.divideBy10();
  /*
   * Assert that values of variables match expectations
   */
  assertEquals(nExpected, n);
  assertEquals(7, k);
}
```

Sometimes, you can write the expected value directly.

Alternative Test Case Style

```
public void testDivideBy10NonZero() {
  /*
   * Set up variables and call method under test
   */
  NaturalNumber n = new NaturalNumber2(1327);
  int k = n.divideBy10();
  /*
   * Assert that values of variables match expectations
   */
  assertEquals("132", n.toString());
  assertEquals(7, k);
}
```

Use `toString`? May be OK, but `equals` is better.

Writing JUnit

- When JUnit is invoked, it will run each of the methods in your test class definition that are marked with the @Test annotation.

```
public class AnimalTest {
        @Test
        public void shouldMoveCorrectly(){
                /* tests here. */
        }
}
```

JUnit 5

The latest release of JUnit uses Java annotations to declare tests.

Annotation	Meaning
@BeforeAll	Executes this method before running any tests.
@AfterAll	Executes this method after running all tests.
@BeforeEach	Executes this method before every test.
@AfterEach	Executes this method after every test.
@Test	Indicates that a method is a test method.

Running JUnit Tests

- To run a JUnit 5 test class from the terminal:

```
java -jar $JUNIT5 -cp . --select-class [test class]

(where $JUNIT5 is the absolulte path to the “standalone” .jar file)
```

- Make sure that the JUnit5 ”standalone” .jar file is in your classpath. Two options:
  - Override your CLASSPATH environment variable.
  - Use the `–cp` switch when invoking java/javac.

Assertions

- The org.junit.jupiter.api.Assertions class is the primary way to test methods.
- A test fails if any of its assertion conditions is not met.
- Examples of assertions:
  - **assertEquals**(Object expected, Object actual);
  - assertEquals(int expected, int actual);
  - assertEquals(double expected, double actual, double tolerance);
  - assertNotNull(Object object);
  - **assertTrue**(boolean condition);
  - assertFalse(boolean condition);

Imports for JUnit

- In the first few lines of any JUnit tester class, you'll want to import both junit and Assertions in the following manner:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.Test;
```

Test Fixtures

- It is a good idea to setup a test fixture in your test class. This is simply a known state that can be starting points for each of your tests.

```
private Animal testAnimal; // instance variable
@BeforeEach
public void setupAnimal {
        //Always start with a brand-new Animal.
        this.testAnimal = new Animal();
}
```

Testing Paradigms

- Implementation-Driven Testing
  - Write your tests classes after/during normal code development.
  - Ensure that the code already written works as it's supposed to.

- Test-Driven Implementation:
  - Write your test classes first.
  - Then start developing your implementation.
  - Use tests to gauge correctness of your code.

END