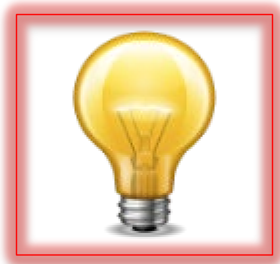
Testing



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

How Big is Big?

- The method bodies we have been writing average maybe a dozen lines of code
- Claim: Boeing 787 Dreamliner avionics (flight control) software has about ...

How Big is Big?

- The method bodies we have been writing average maybe a dozen lines of code
- Claim: Boeing 787 Dreamliner avionics (flight control) software has about ***6.5 million*** lines of code
- Claim: Microsoft Windows 10 has about ...

How Big is Big?

- The method bodies we have been writing average maybe a dozen lines of code
- Claim: Boeing 787 Dreamliner avionics (flight control) software has about ***6.5 million*** lines of code
- Claim: Microsoft Windows 10 has about ***50 million*** lines of code
- Claim: a modern car has about ...

How Big is Big?

- The method bodies we have been writing average maybe a dozen lines of code
- Claim: Boeing 787 Dreamliner avionics (flight control) software has about ***6.5 million*** lines of code
- Claim: Microsoft Windows 10 has about ***50 million*** lines of code
- Claim: a modern car has about ***100 million*** lines of code (though this figure is highly dubious)

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***

Unit Testing:

This is the kind of testing we will do in this course and the next.

- **Best practice** is to te ividual ***units*** or ***components*** of softw e (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***

Unit Testing:

> The unit being tested is known as the ***UUT***, or ***unit under test***.

- **Best practice** is to te ividual ***units*** or ***components*** of softw e (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***

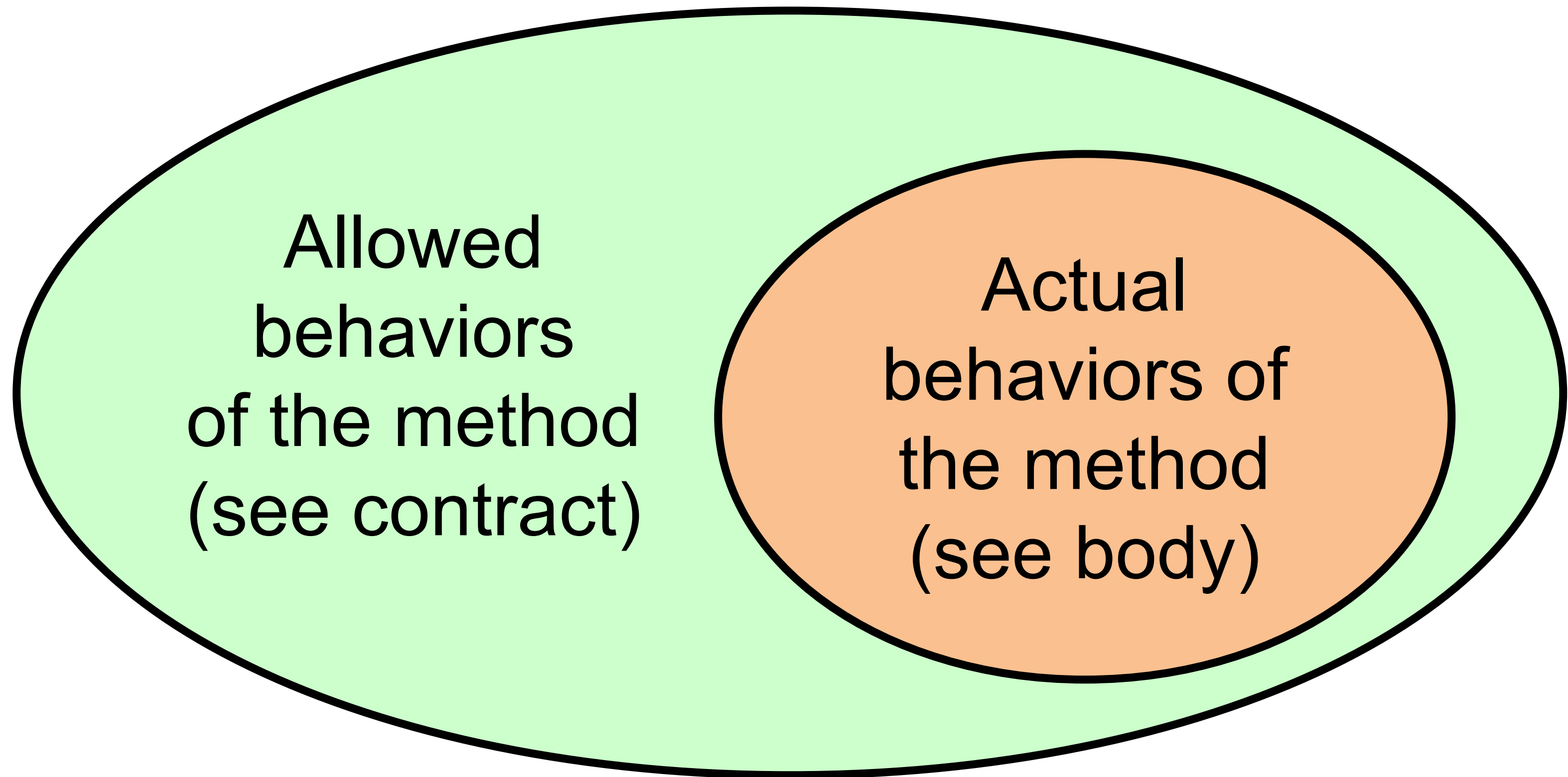
Testing Functional Correctness

- 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



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) {...}
```

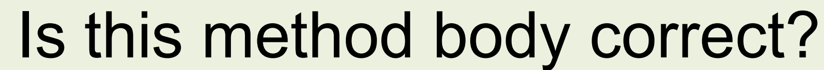
This means:
"*n* is divisible by *aFactor*".

Example Method Body

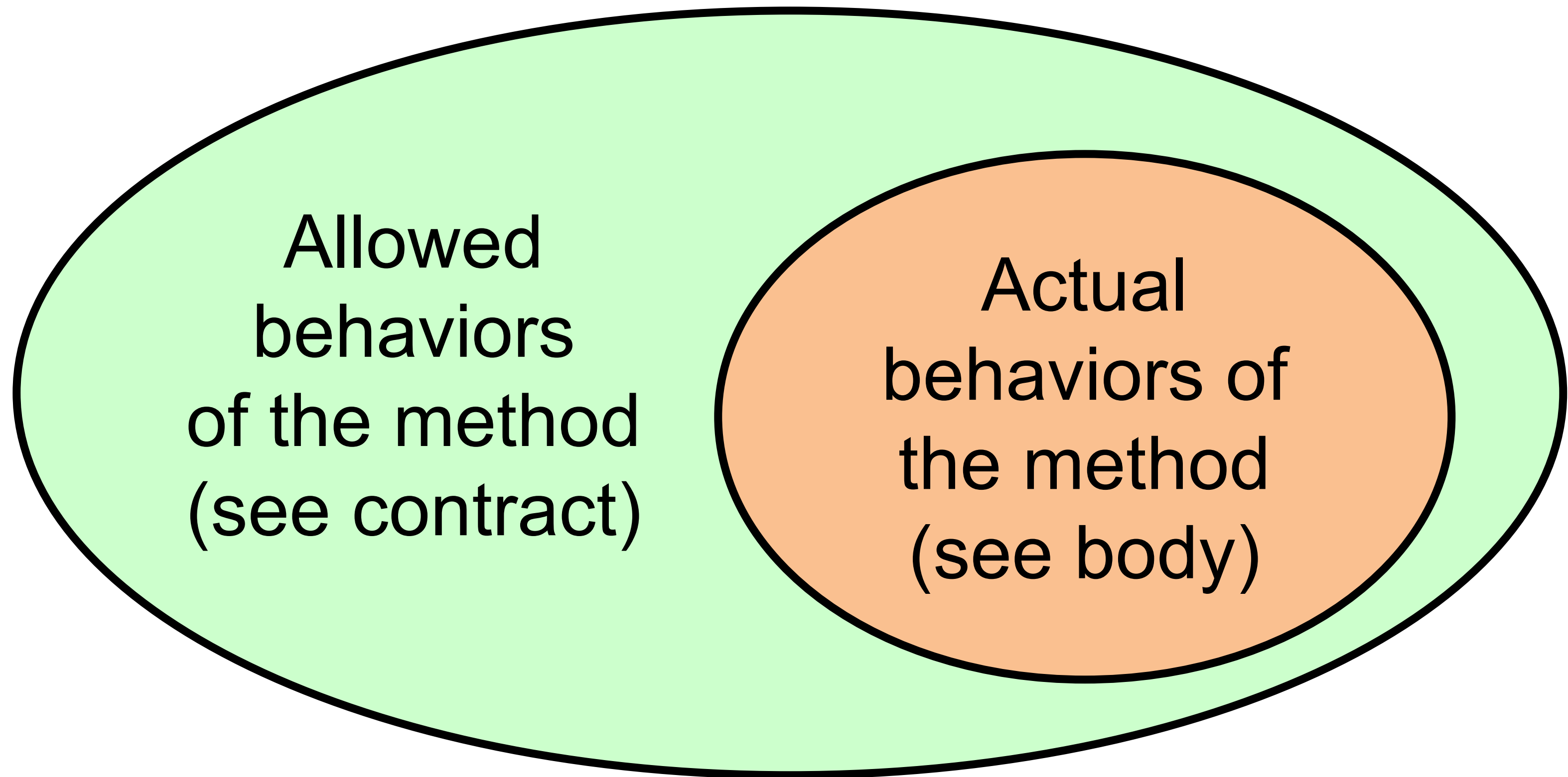
```
private static int aFactor(int n) {
  return 1;
}
```

Example Method Body

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



Behaviors



...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`

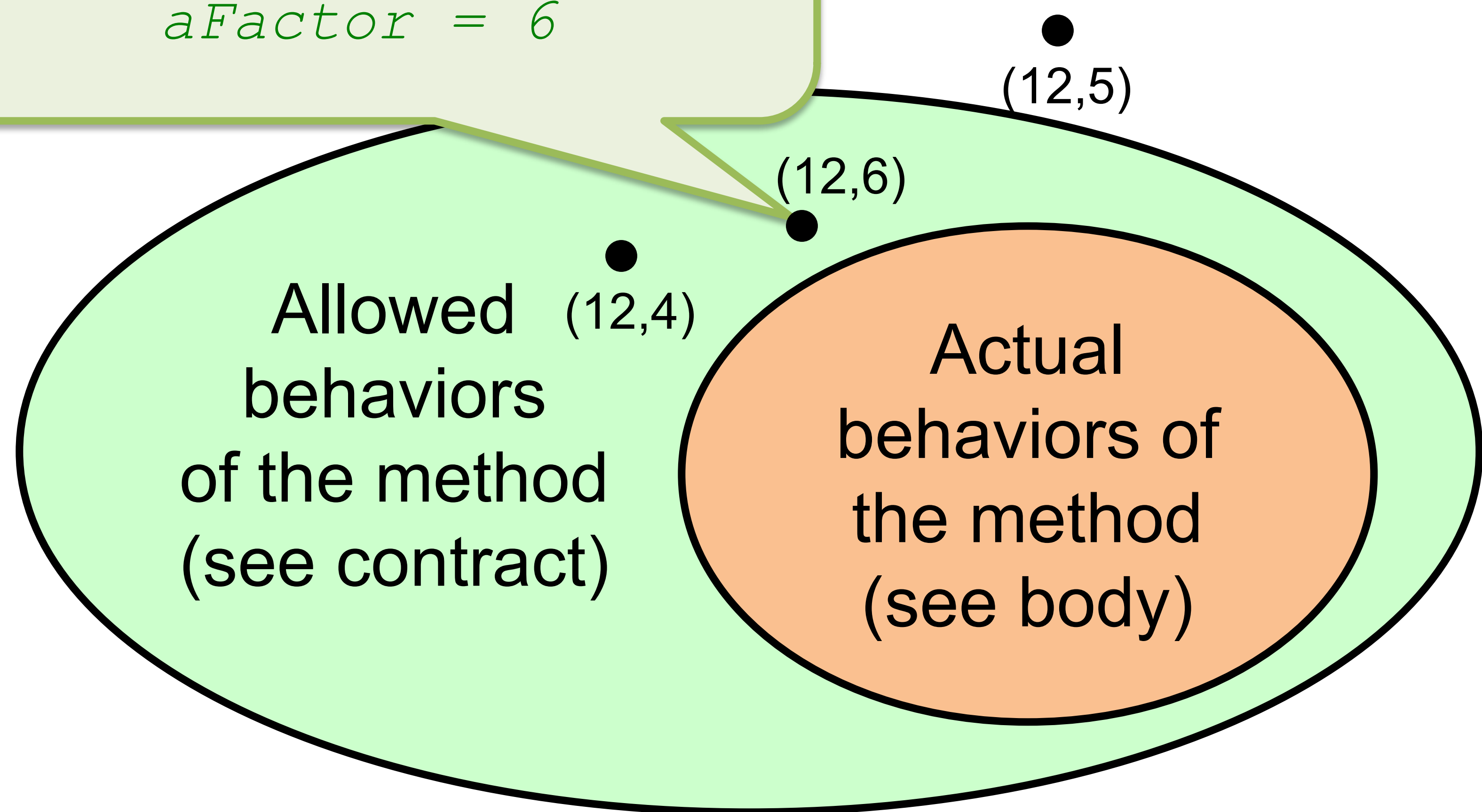
`aFactor = 5`

(12,5)

Allowed behaviors of the method (see contract)

(12,4)

Actual behaviors of the method (see body)



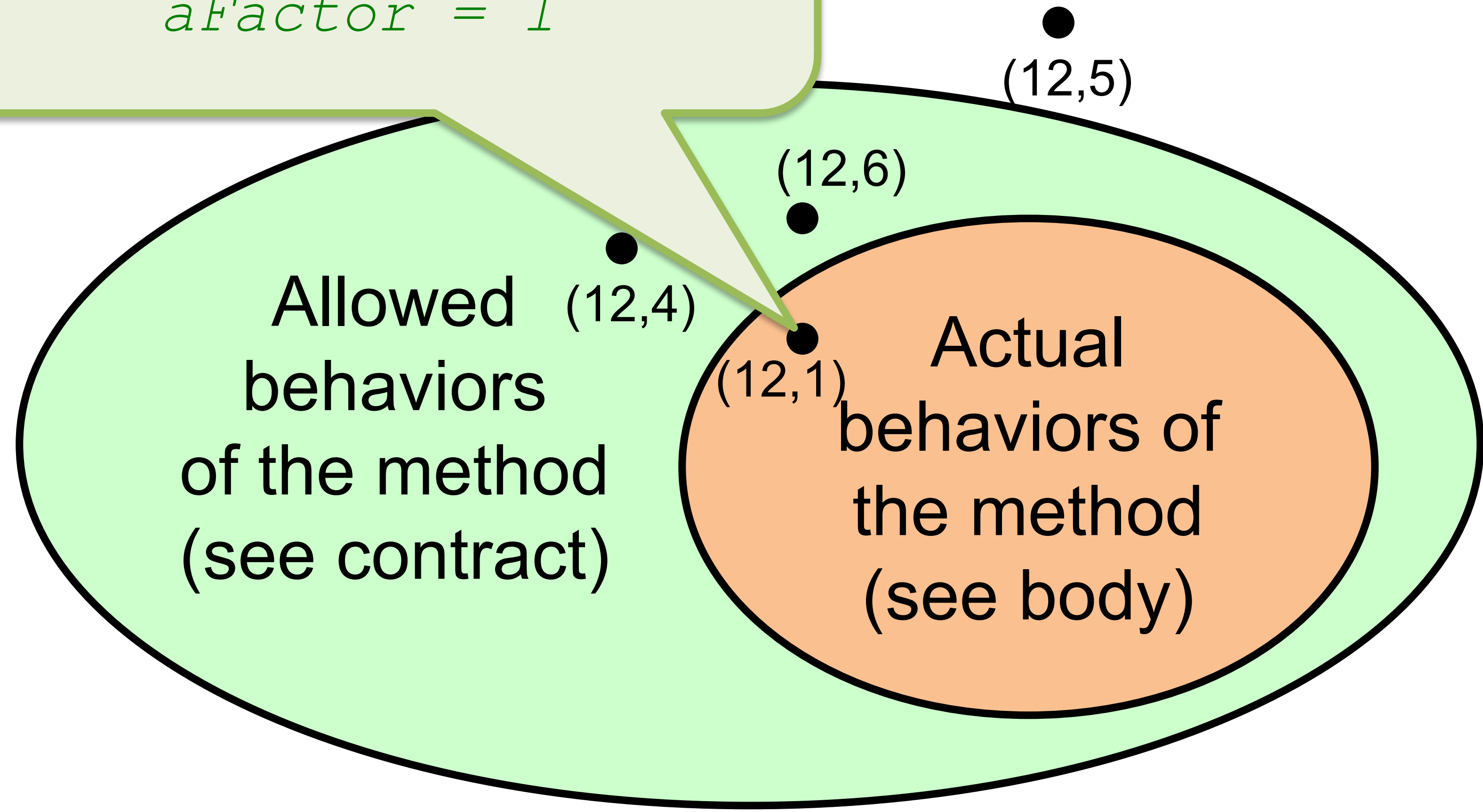
Contract for aFactor allows:
n = 12
aFactor = 6
iors
(12,5)
(12,6)
(12,4)
Allowed behaviors of the method (see contract)
Actual behaviors of the method (see body)

iors

Contract for `aFactor` allows:

`n = 12`

`aFactor = 1`

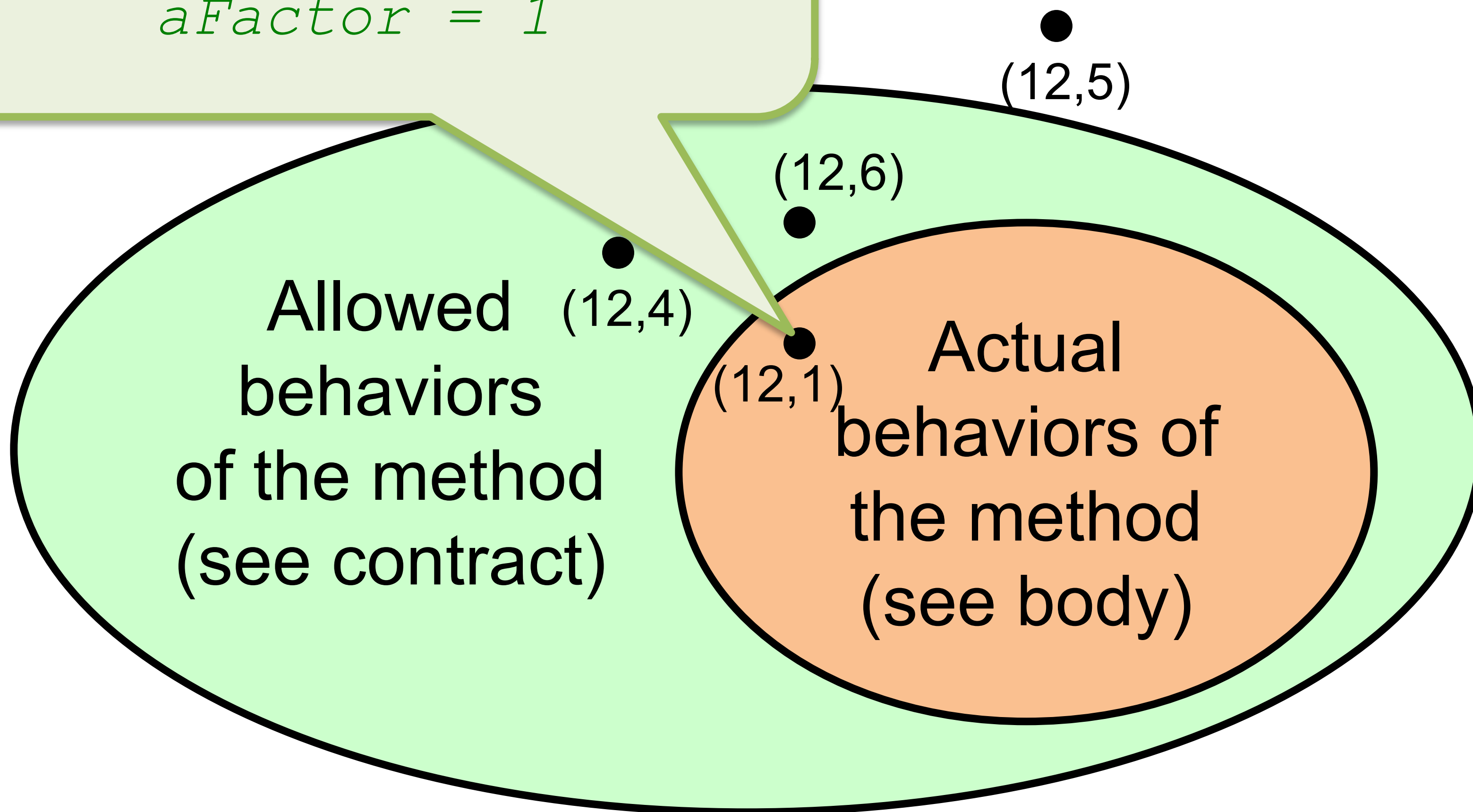


iors

Body for `aFactor` gives:

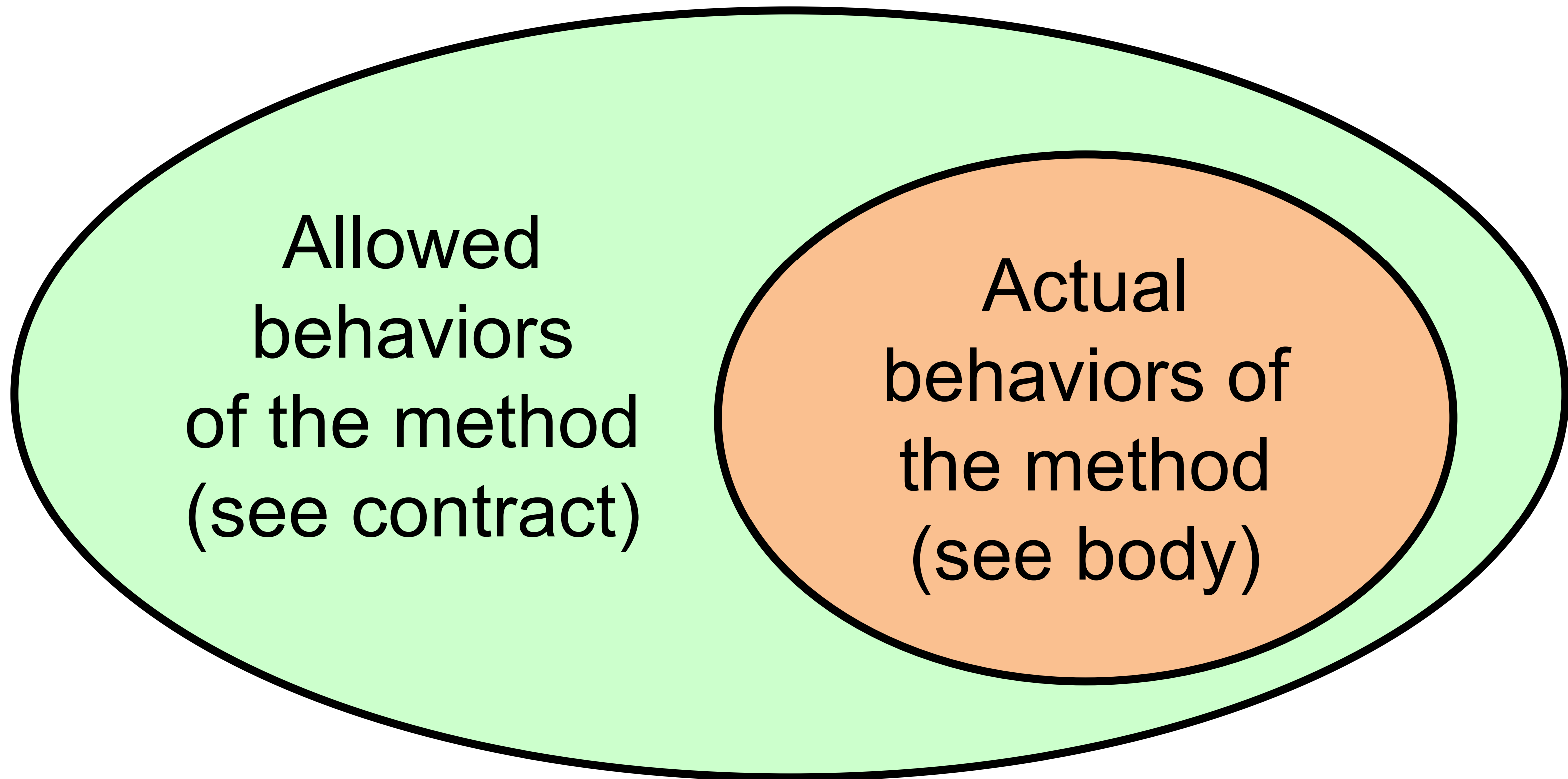
n = 12

aFactor = 1



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 *is* … e 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

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

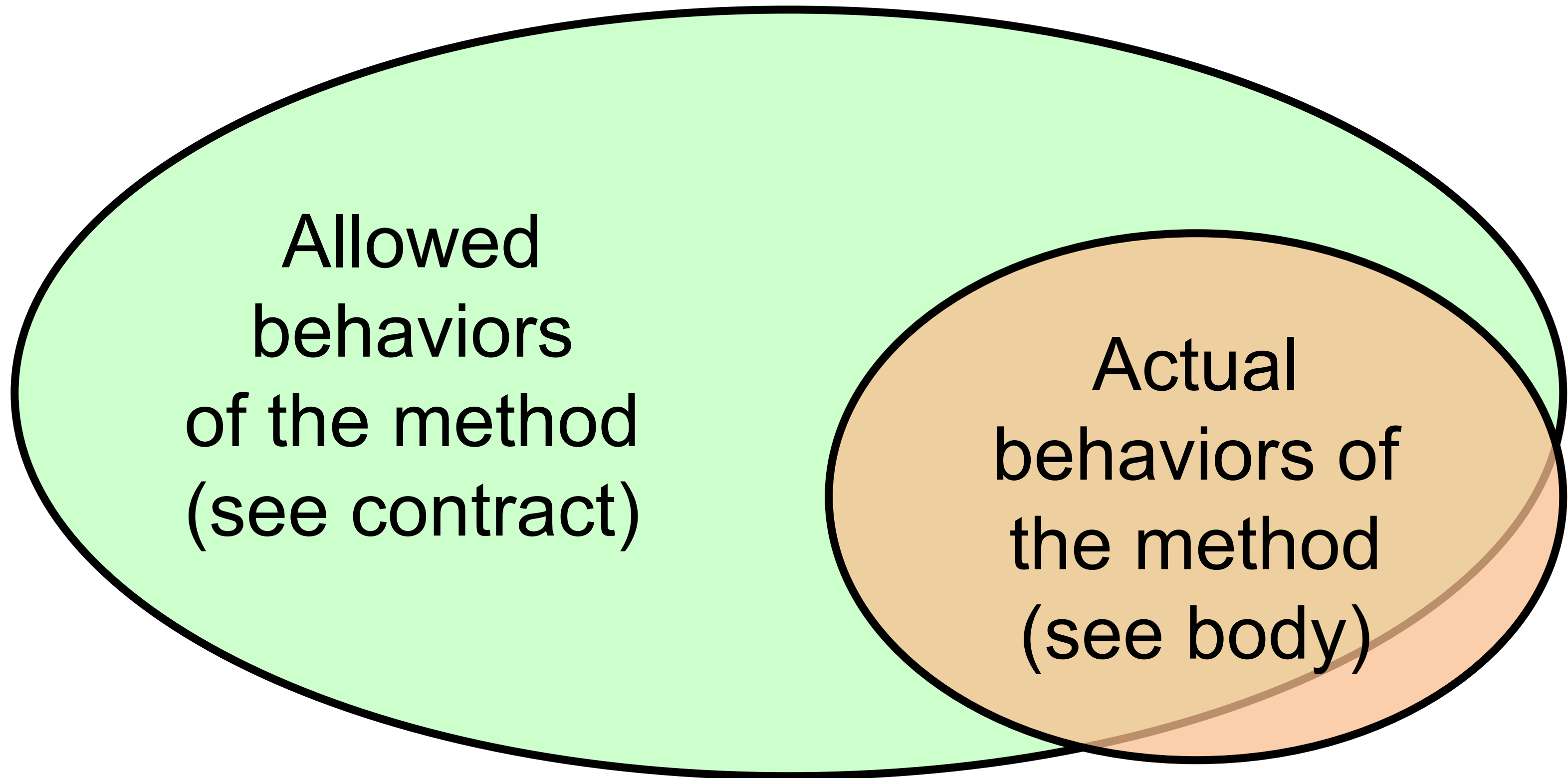
- Design and coding are ***creative*** activities
- Testing is a ***destructive*** activity
  - The primary goal is to "break" the software, i.e., to show that it has defects
- Very often the same person does both coding and testing (*not* a **best practice**)
  - You need a "split personality": when 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

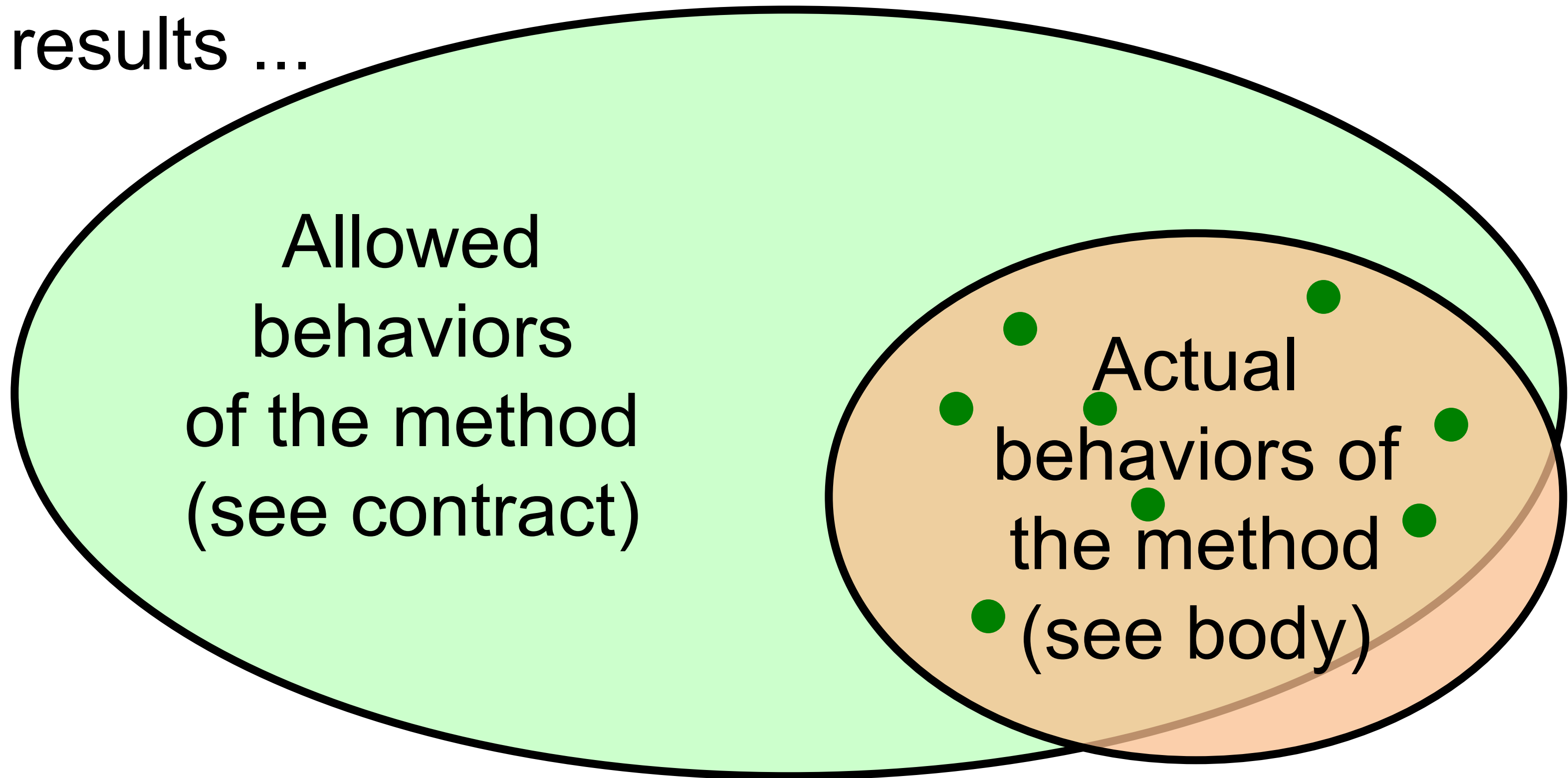
Incorrect (Defective) Code

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



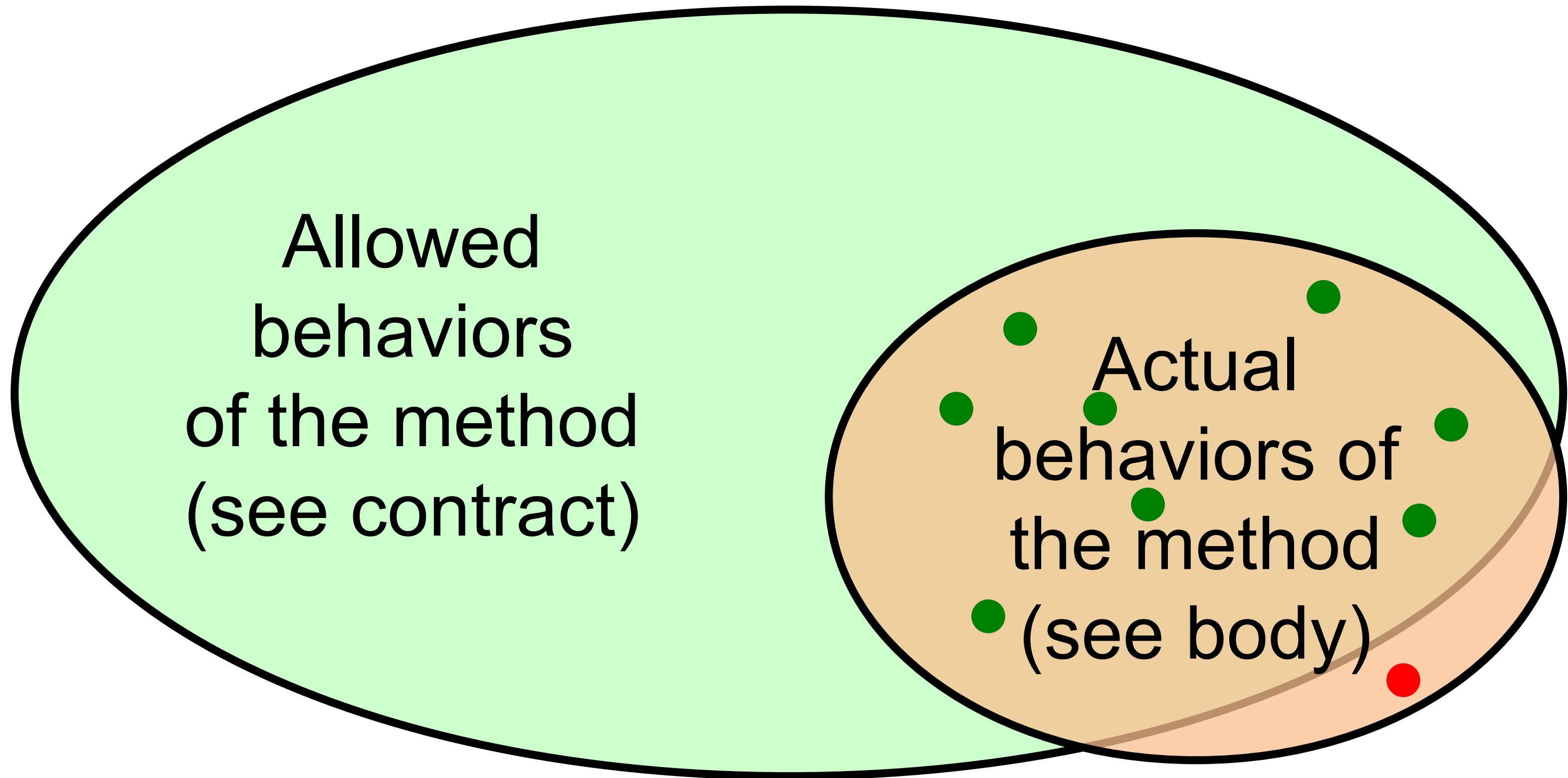
Incorrect (Defective) Code

- ... 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!

If this happens, testing has ***succeeded*** (in *revealing a defect* in the method body).

of the meth
(see contract)

Actual
behaviors of
the method
(see dy)

Test Cases

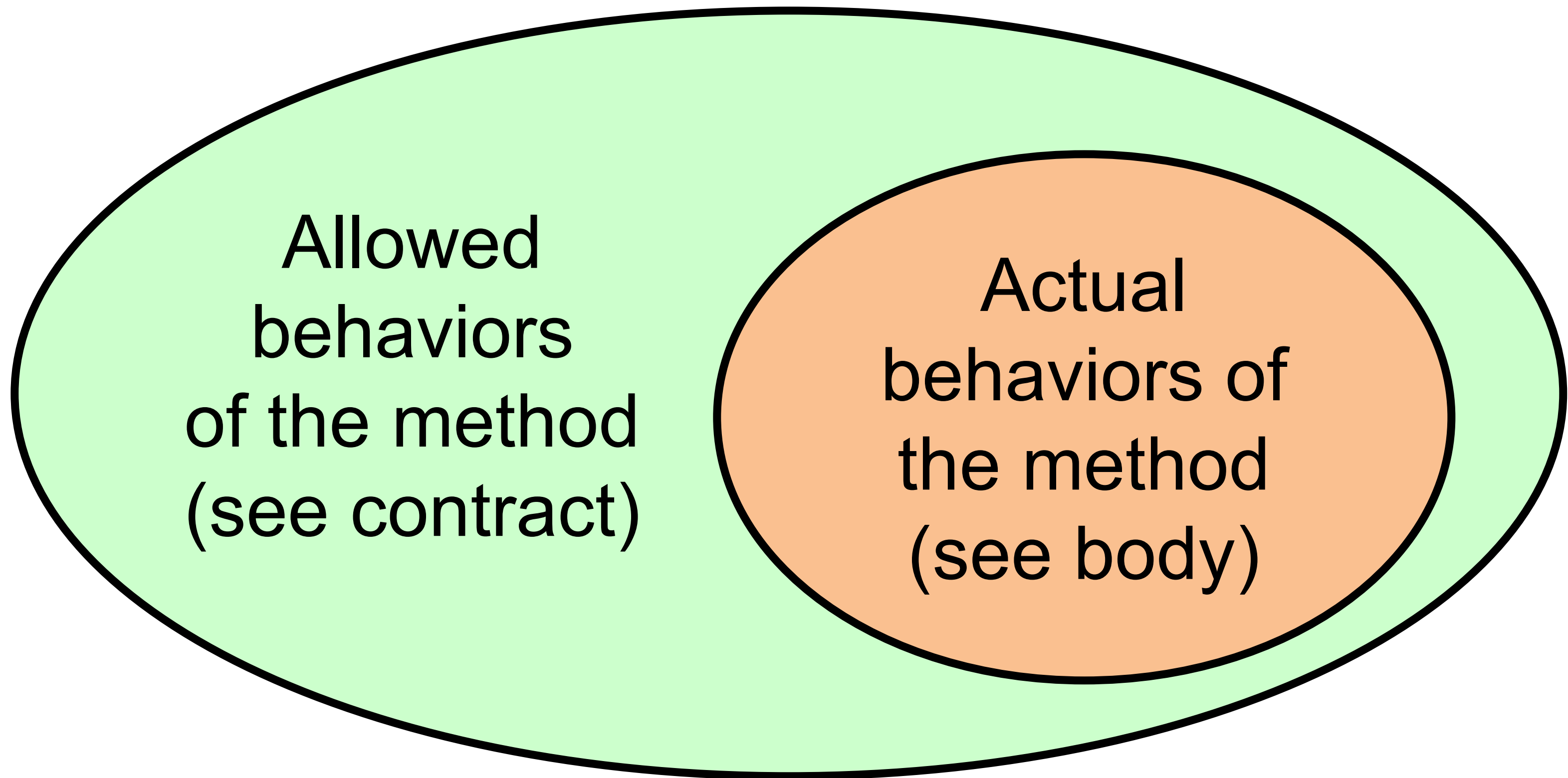
- Each input value and corresponding allowed/expected result is a ***test case***
- Test cases that do *not* reveal a defect in the code do not help us refute a claim of correctness
- Test cases like that last one should be cherished!

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

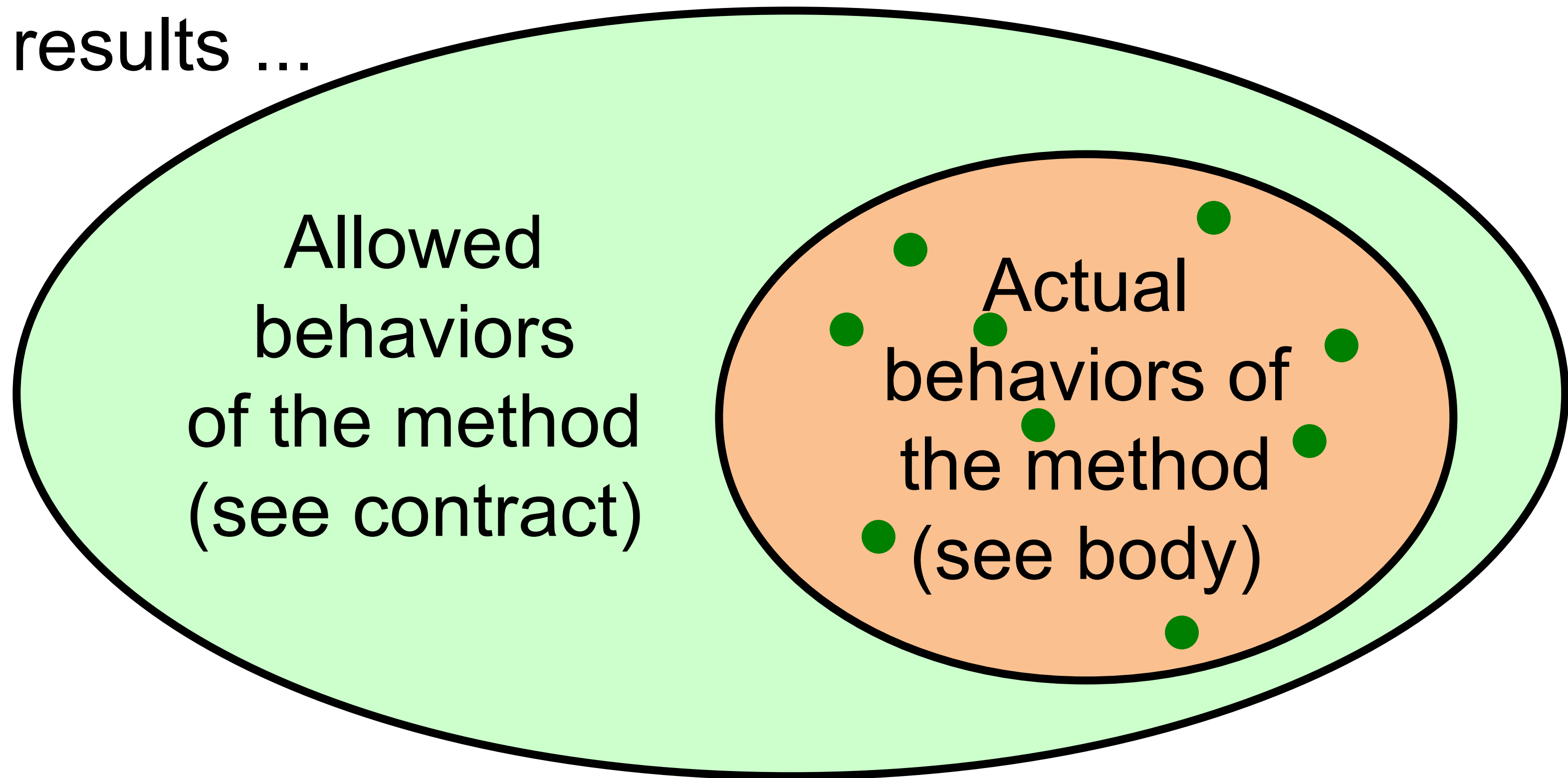
Correct Code

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



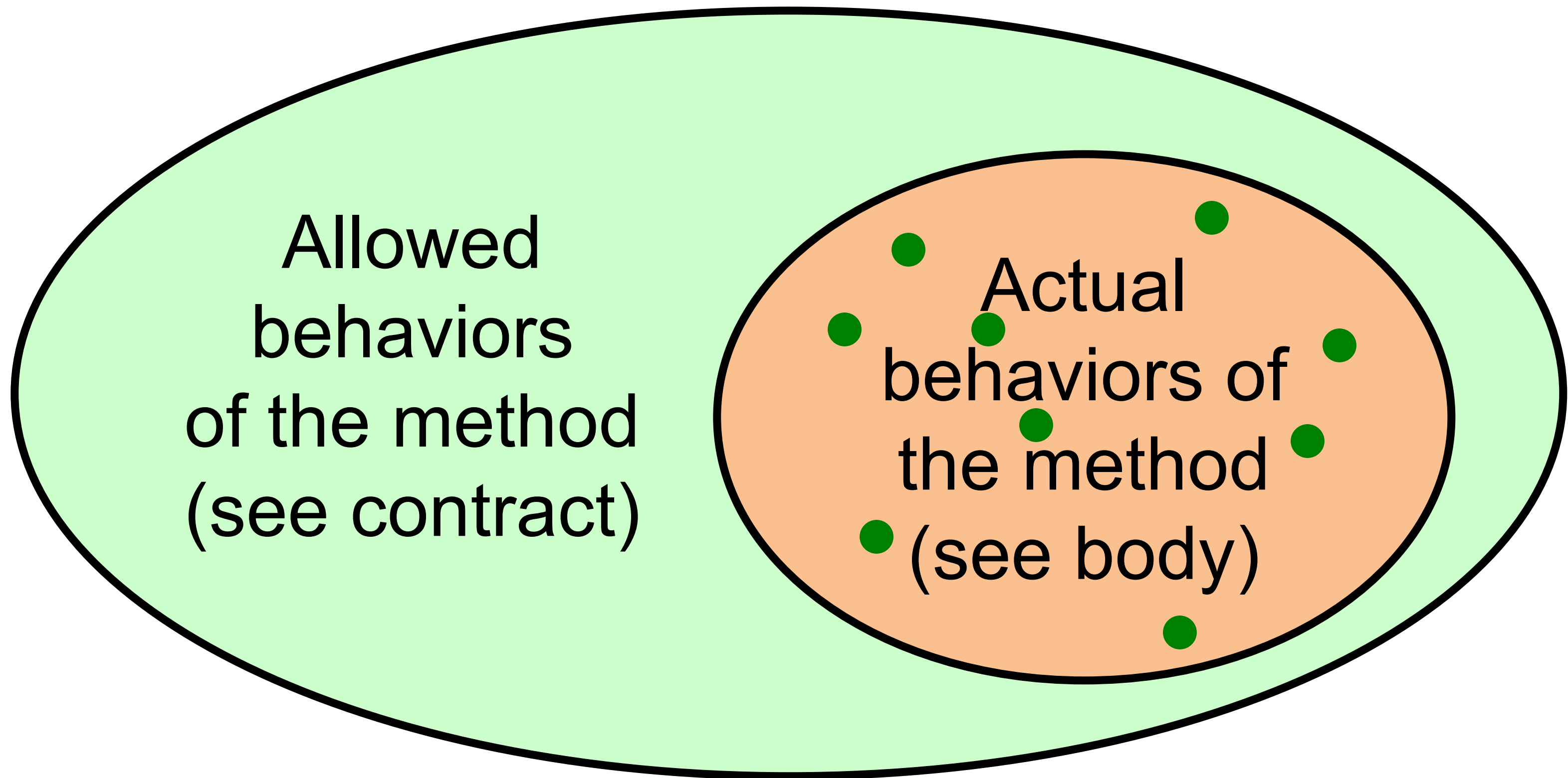
Correct Code

- ... 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?

Resources

- *Software Testing* (Brian Hambling, *et al*., 2010)
  – https://library.ohio-state.edu/record=b8532947~S7