# Unit Testing Methods

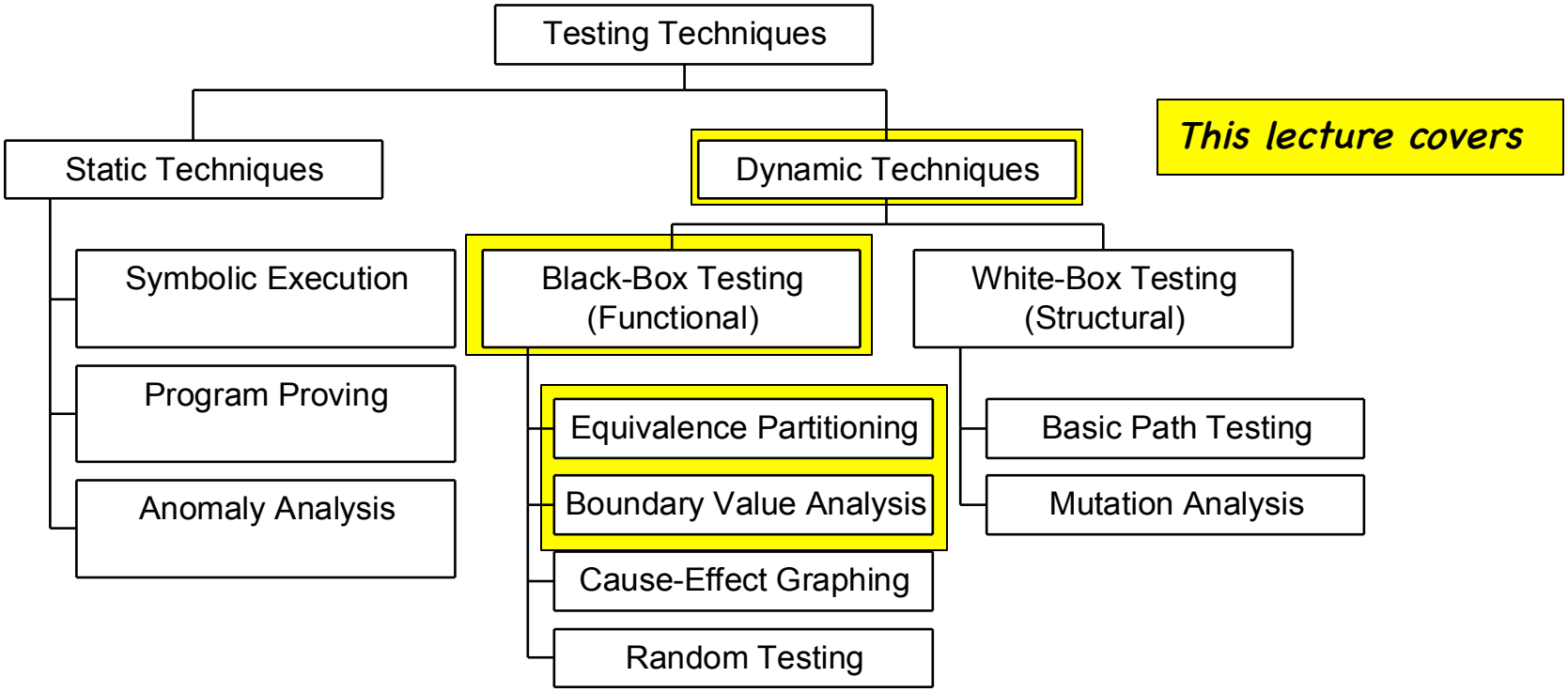
Black-box Testing

Extensions

# Unit Testing Techniques

Unit Testing checks that an individual program unit (subprogram, object class, package, module) behaves correctly.

- Static Testing - testing a unit without executing the unit code
- Dynamic Testing - testing a unit by executing a program unit using test data

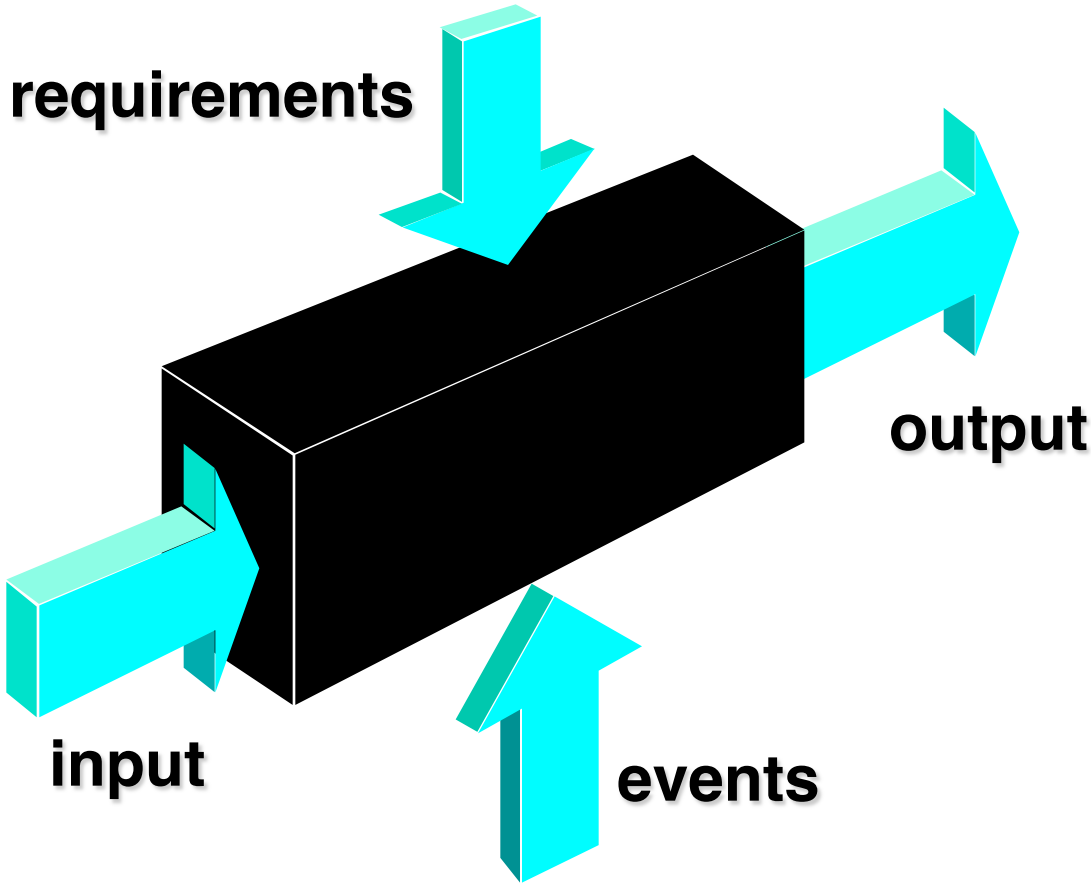



***Program testing can be used to show the presence of bugs, but never to show their absence [Dijkstra]***

# Black-Box Testing

Black box testing

- Purpose: tests the overall functional behavior
- Drawback: may not reveal cause of the defect
- Method: Treat the module as a classic input/output module; when you provide it with certain inputs it should provide certain outputs

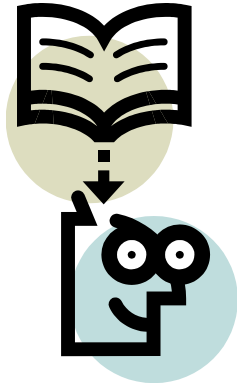


# Specification-Based Testing

Black-box Testing is known as *specification-based testing*

- Test cases designed, selected, and run based on specifications

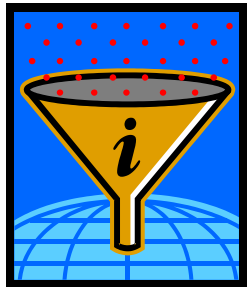
Use specifications to derive test cases



- Requirements
- Design
- **Function signature**

Based on some kind of input domain

- The approach is not naïve, but considers the range of data inputs
  - Typical values
  - Boundary values
  - Special cases
  - Invalid input values (negative testing)



# Black-box Testing (naïve approach)

*Guideline-based testing*:

- Review documentation for expected output
- Example: “A Fibonacci sequence starts like this:
  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89”

Unit Test Procedure:

- Ask the user to provide a number from 0 to 11 inclusive
- Call the Fib function
- Compare result value to the corresponding value in the above list
- *Question: What test input sets do you think are needed?*

***We say this is naïve because it only considers the main use (success) case with no special or negative cases***

Alternative: *Partition-based testing*

- In this paradigm, we attempt to identify classes of inputs and outputs

# Equivalence Partitioning

*Equivalence partitioning* is an approach to black box testing that *divides* the input domain of a program into *classes of data* from which test cases can be derived.

Discrete Math review: *What is an equivalence class?*

*Equivalence classes* are based on *equivalence relations*:

*For all x, y, z in set T, the following hold:*

- *Reflexivity: x ~ x*
- *Symmetric: x ~ y iff y ~x*
- *Transitivity: if x ~ y and y ~ z, then x ~ z*

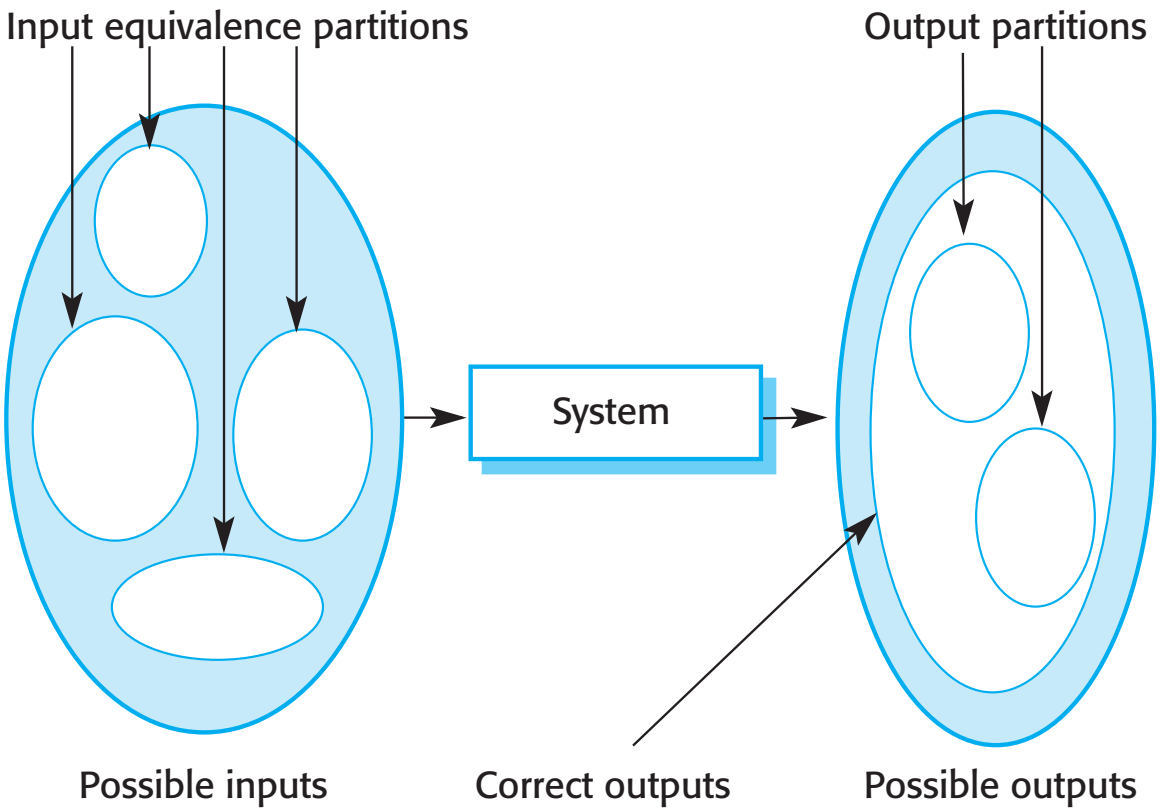
  Examples: "=" "has the same astrological sign"

  Non-examples: ">" "is taller than"

**How does this help?**

- Define *equivalence* as "these data are expected to produce the same result"
- We only need to test with 1 representative data element from that class

# Equivalence partitioning



# Equivalence partitioning

Example: Suppose a program computes the value of the function $\sqrt{(X-1)*(X+2)}$. What are the equivalence classes?

- X < = -2 valid
- -2 < X < 1 invalid
- X >= 1 valid

Test cases should be selected from each equivalence class.

- Cons:
  - Lacks causality: 2 inputs may expect to produce the same output for entirely different reasons
  - No deterministic way to define the classes; often difficult for non-mathematical operations or multi-dimensional inputs
- Pros:
  - Often one can intuitively identify representative data, so there really is no reason not to try this.
  - Can greatly reduce the number of test cases you need to run

# Boundary Value Analysis

*Boundary Value Analysis* - black box technique where test cases are designed to test the boundary of an input domain.

- Studies have shown that more errors occur on the "boundary" of an input domain than in the "center".
- Commonly referred to as a type of “edge case”

Boundary value analysis complements and can be used in conjunction with equivalence partitioning.

Example:

After using equivalence partitioning to generate equivalence classes, boundary value analysis would dictate that the boundary values of the three ranges be included in the test data. That is, we might choose the following test cases (for a 32 bit system):

| | |
|---|---|
| X <= -2 | $-2^{31}$, -100, -2.1, -2 |
| -2 < X < 1 | -1.9, -1, 0, 0.9 |
| X >= 1 | 1, 1.1,100, $2^{31}-1$ |

# Example: Possible Bases

**Consider this simple function:**

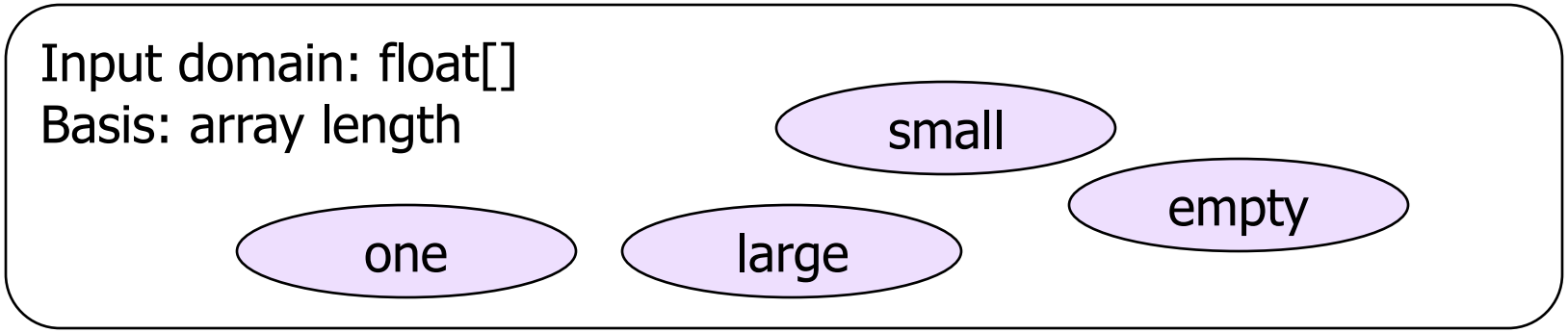
```
1 float homeworkAverage(float[] scores) {
2    float min = 99999;
3    float total = 0;
4    for (int i=0; i < scores.length ; i++) {
5       if (scores[i] < min)
6          min = scores[i];
7       total += scores[i];
8    }
9    total = total - min;
10   return total / (scores.length - 1);
11 }
```

# Example: Possible Bases

**Basis: *Array length***

- Null array
- Empty array
- 1 element
- 2 or 3 elements
- Lots of elements
- Scores.length elements

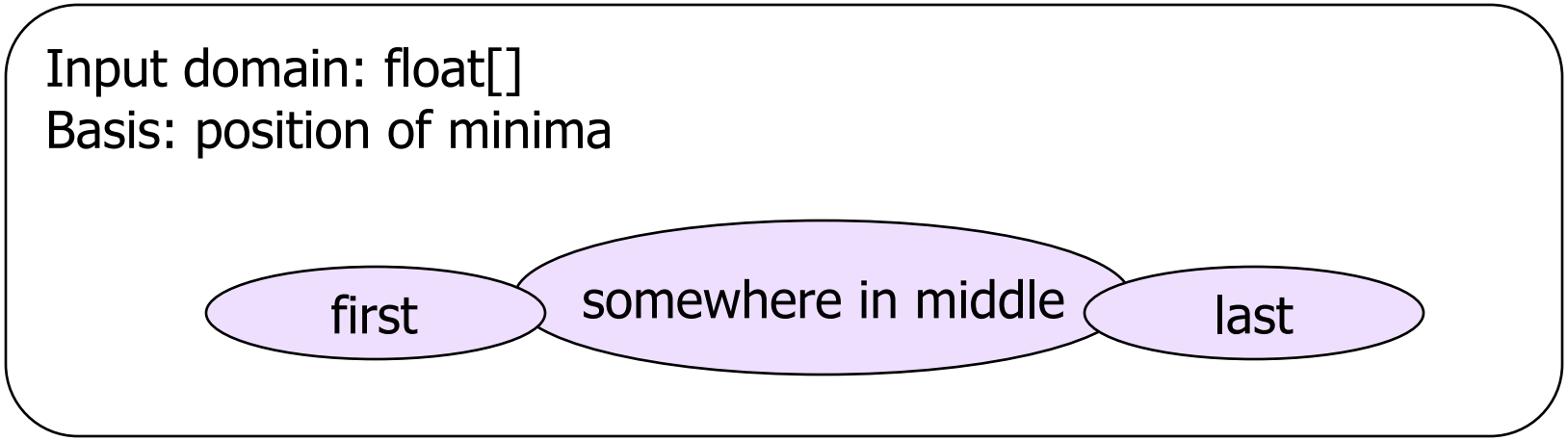
```
1  float homeworkAverage(float[] scores) {
2    float min = 99999;
3    float total = 0;
4    for (int i = 0 ; i < scores.length ; i++) {
5      if (scores[i] < min)
6        min = scores[i];
7      total += scores[i];
8    }
9    total = total – min;
10   return total / (scores.length – 1);
11 }
```



# Example: Possible Bases

**Basis: *Position of minimum score***

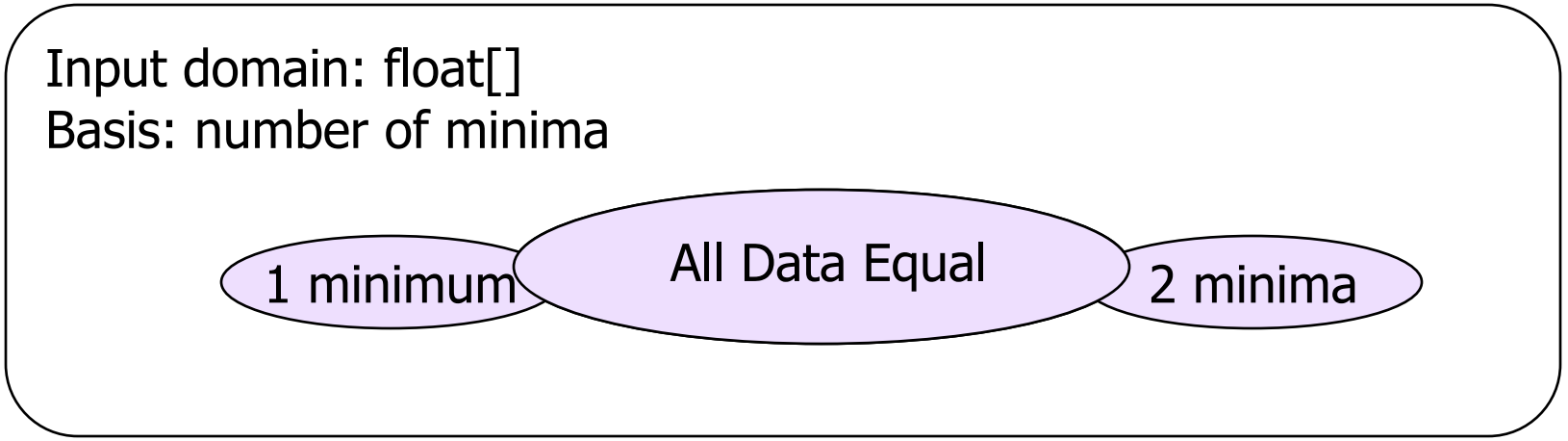
- Smallest element first
- Smallest element in middle
- Smallest element last



# Example: Possible Bases

**Basis: *Number of minima***

- Unique minimum
- A few minima
- All minima



# Wrap-up Discussion Problem

Which of these 2 methods requires more tests?

Which of the 2 is *easier* to construct bbox tests for?

```
// method1 – determines if a string is a palindrome
// returns: boolean
// params: a String
// exception – if parameter isn't a string (has whitespace)
public boolean method1(String s) throws Exception

// method2 – prints the value of an number in base b to an int
// returns – void
// params: String which must represent a number in base of 2nd param
// exceptions: None
public void method2(String s, int b)
```

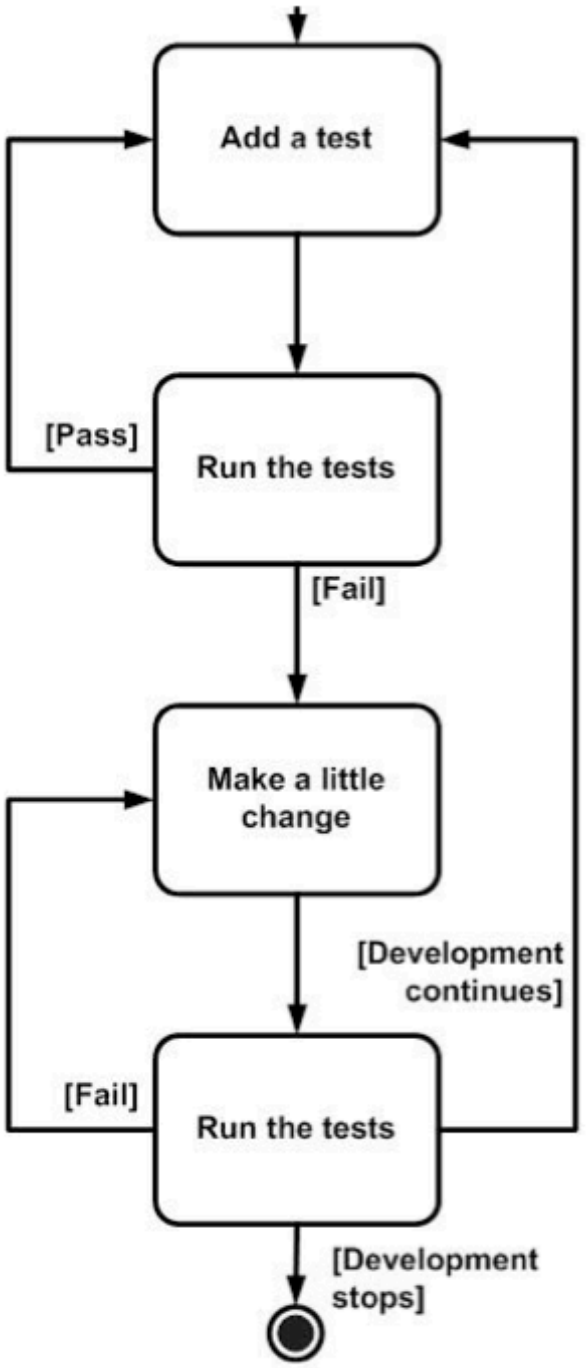
# TDD: An Alternative View

## Test-Driven Development (TDD)

- Write your tests first!
- A promoted practice for XP
  - “code a little, test a little” becomes “test a little, code a little”
- Derives from your user story acceptance tests

## Also called “Red-Green-Refactor”

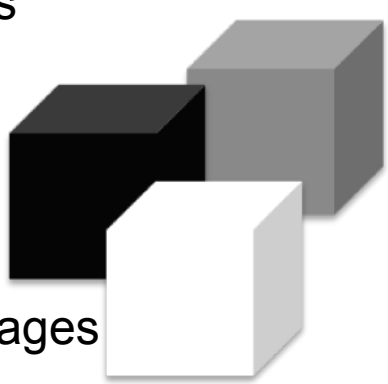
- Idea is, if you write the test to the specification, then the only thing in your implementation will be that which satisfies the test.
- “Lightweight” as it adds no extraneous behaviors!



# Unit Testing: OO Perspective

So far we have assumed our units are discrete functions

- But we can also decompose other ways, including using objects



1. Approach 1: Focus on Interactions
   - Create a unit test per method
     - But this is not considered a *best practice* as methods represent conceptual *behaviors* and as such may not be used in isolation.
   - Unit test a *scenario* as a collection of objects exchanging messages
     - Think UML sequence diagram
     - Use-case driven unit tests often happen at this level
   - Sommerville suggests doing interface testing
     - This is OK but is more in the category of *integration testing* to me
2. Approach 2: Focus on Behaviors
   - Consider the object or component itself as the unit, not the scenario
     - Then your guide is the *statechart*, as it defines the valid set of states an object or component can go through
     - Usually this is associated with *white box unit testing of objects*

***What is the gray box over there? Gray-box testing is an emerging concept where the tester may have some knowledge of internal structures but not of the code directly. Sometimes used in security***

# Unit Tests: Who does it?

*Developer writes the tests* of the code s/he just produced

- Needs to ensure that the code functions properly before releasing it to the other developers

Benefits: knows the code best

Drawback: knows the code best

- Bias - “I trust my code”; “I always write correct code”
- Blind spots - The same developer who didn’t see “gaps” in her/his code won’t see “gaps” in her/his unit test cases

*Someone else writes the tests* of the code just produced

- The “get another set of eyeballs on it” approach

Benefits: does not know the code already

- Lack of bias, creates familiarity with the code

Drawback: does not know the code already

- So it is going to take longer to develop the tests
- Does not support TDD