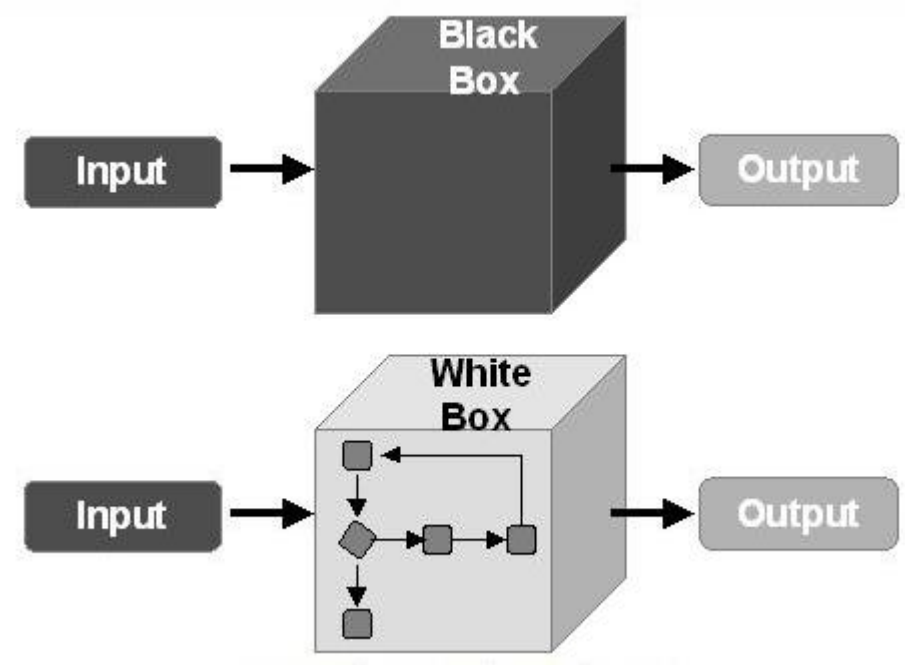
EN.601.422 / EN.601.622

Software Testing & Debugging

Whitebox vs Blackbox Testing



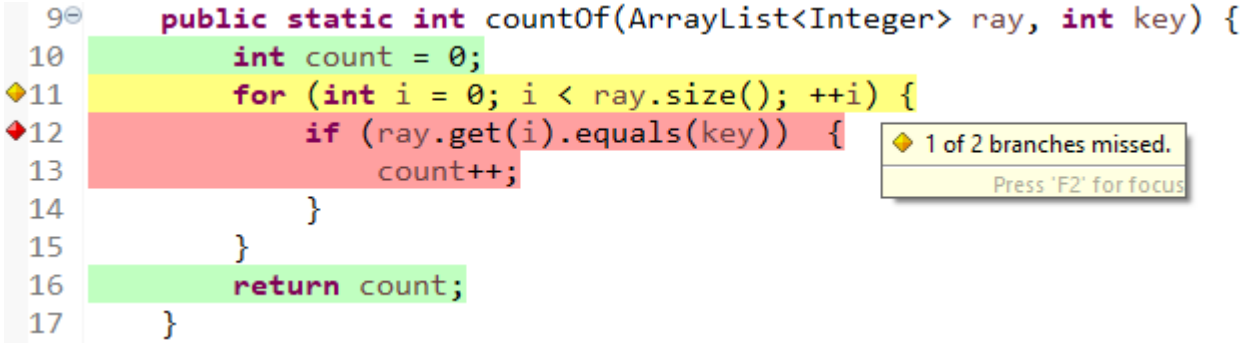
Whitebox Testing

- An *internal* perspective
- Testing based on source code
  - Choose inputs to exercise *paths* through the code
- Also known as *glass-box, transparent or structural* testing

Whitebox Coverage Criteria

- **Method Coverage (MC):** each and every method has been called at least once
- **Statement Coverage (SC):** all statements in a method have been executed at least once
- **Branch Coverage (BC):** each and every possible branch from each decision point is executed at least once
- **Path Coverage (PC):** All possible execution paths are executed at least one

Example




```
// Test
ArrayList<Integer> ray = new ArrayList<Integer>();
countOf(ray, 2);
```

only achieves MC

Example

```
 9    public static int countOf(ArrayList<Integer> ray, int key) {
10        int count = 0;
11        for (int i = 0; i < ray.size(); ++i) {
12            if (ray.get(i).equals(key))  {
13                count++;
14            }
15        }
16        return count;
17    }
```

```
// Test
ArrayList<Integer> ray = new ArrayList<Integer>();
ray.add(2);
countOf(ray, 2);
```

achieves MC and SC

how about BC?

Example

```
 9    public static int countOf(ArrayList<Integer> ray, int key) {
10        int count = 0;
11        for (int i = 0; i < ray.size(); ++i) {
12            if (ray.get(i).equals(key))  {
13                count++;
14            }
15        }
16        return count;
17    }
```

```
// Test
ArrayList<Integer> ray = new ArrayList<Integer>();
ray.add(1);
ray.add(2);
countOf(ray, 2);
```

achieves MC, SC and BC

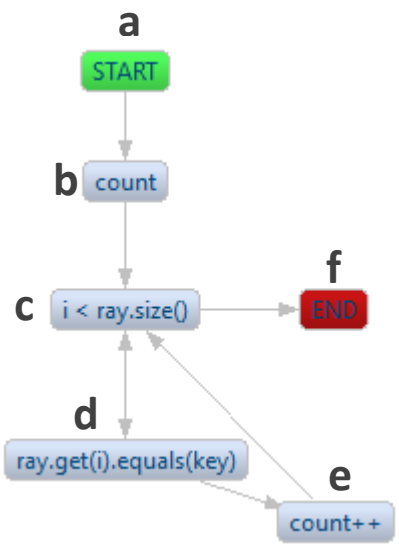
how about PC?

Whitebox Coverage Criteria: Path Coverage

- In general, not possible to achieve full path coverage
  - programs contain loops
  - some paths might be infeasible
- Only possible to achieve PC up to a certain depth in a loop
  - possible to achieve full path coverage in programs without loops (or with only hard-bounded loops)

Path Coverage (up to depth 1)

```
public static int countOf(ArrayList<Integer> ray, int key) {
   int count = 0;
   for (int i = 0; i < ray.size(); ++i) {
      if (ray.get(i).equals(key))  {
         count++;
      }
   }
   return count;
}
// Tests
List ray0 = new ArrayList<Integer>();
List ray1 = new ArrayList<Integer>();
ray1.add(2);
countOf(ray0, 2); // abcf
countOf(ray1, 1); // abcdcf
countOf(ray1, 2); // abcdecf
```



Paths to cover:

abcf
abcdcf
abcdecf

How to Deal with Loops?

- Testing all possible iterations of loops can be impractical/unscalable

Loop Boundary Adequacy

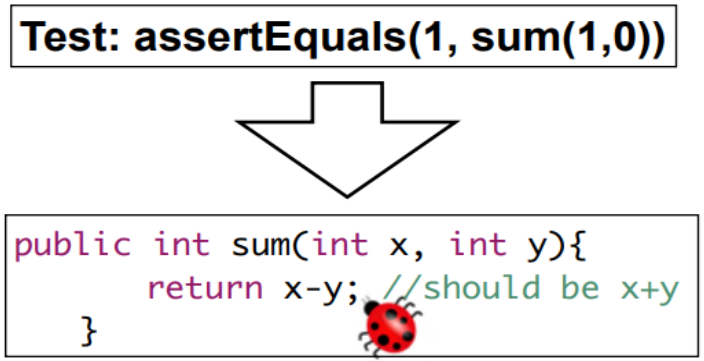
- A test suite satisfies the loop boundary adequacy criterion if for every loop **L**:
  1. There is a test case which iterates L **zero** times
  2. There is a test case which iterates L **once**
  3. There is a test case which iterates L **more than once**

Loop Boundary Adequacy

- Loop Boundary Adequacy is usually combined with other adequacy criteria such as SC, BC, etc.

Whitebox Coverage Criteria

- 100% coverage is never a guarantee of fault-free software



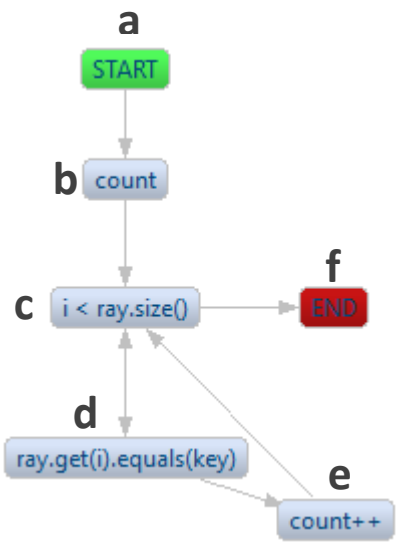
achieves 100% SC, BC and PC

- Research has shown though that in general:
  - higher code coverage ➔ more faults revealed

Example

- Let's write tests to achieve path coverage of up to depth 2 for the *countOf* method:

```
public static int countOf(ArrayList<Integer> ray, int key) {
   int count = 0;
   for (int i = 0; i < ray.size(); ++i) {
      if (ray.get(i).equals(key))  {
         count++;
      }
   }
   return count;
}
```



Paths to cover

abcf
abcdcf
abcdecf
abcdcdcf
abcdcdecf
abcdecdecf
abcdecdcf

The Test Class for ArrayUtils

```
public class ArrayUtilsWbTest {
  @Test // abcf
  public void testCountOfEmptyArr() {
    ArrayList<Integer> ray = new ArrayList<Integer>();
    assertTrue(ArrayUtils.countOf(ray, 2) == 0);
  }
  @Test // abcdcf
  public void testCountOfArrSizeOneKeyNotExists() {
    ArrayList<Integer> ray = new ArrayList<Integer>();
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 1), 0);
  }
  @Test // abcdecf
  public void testCountOfArrSizeOneKeyExists() {
    ArrayList<Integer> ray = new ArrayList<Integer>();
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 2), 1);
  }
  @Test // abcdcdcf
  public void testCountOfArrSizeTwoKeyNotExists() {
    ArrayList<Integer> ray = new ArrayList<Integer>();
    ray.add(1);
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 3), 0);
  }
  @Test // abcdecdcf
  public void testCountOfArrSizeTwoKeyExistsFirst() {
    ArrayList<Integer> ray = new ArrayList<Integer>();
    ray.add(1);
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 1), 1);
  }
  @Test // abcdcdecf
  public void testCountOfArrSizeTwoKeyExistsSecond() {
    ArrayList<Integer> ray = new ArrayList<Integer>();
    ray.add(1);
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 2), 1);
  }
  @Test // abcdecdecf
  public void testCountOfArrSizeTwoKeyExistsFirstSecond() {
    ArrayList<Integer> ray = new ArrayList<Integer>();
    ray.add(1);
    ray.add(1);
    assertEquals(ArrayUtils.countOf(ray, 1), 2);
  }

} end of class ArrayUtilsWbTest
```

Utilizing Test Fixture

```
public class ArrayUtilsWbTest {
  ArrayList<Integer> ray;
  @BeforeEach
  public void setup() {
    ray = new ArrayList<Integer>();
  }
  @Test // abcf
  public void testCountOfEmptyArr() {
    assertTrue(ArrayUtils.countOf(ray, 2) == 0);
  }
  @Test // abcdcf
  public void testCountOfArrSizeOneKeyNotExists() {
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 1), 0);
  }
  @Test // abcdecf
  public void testCountOfArrSizeOneKeyExists() {
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 2), 1);
  }
  @Test // abcdcdcf
  public void testCountOfArrSizeTwoKeyNotExists() {
    ray.add(1);
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 3), 0);
  }
  @Test // abcdecdcf
  public void testCountOfArrSizeTwoKeyExistsFirst() {
    ray.add(1);
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 1), 1);
  }
  @Test // abcdcdecf
  public void testCountOfArrSizeTwoKeyExistsSecond() {
    ray.add(1);
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 2), 1);
  }
  @Test // abcdecdecf
  public void testCountOfArrSizeTwoKeyExistsFirstSecond() {
    ray.add(1);
    ray.add(1);
    assertEquals(ArrayUtils.countOf(ray, 1), 2);
  }
} end of class ArrayUtilsWbTest
```

Putting it all together: Blackbox Testing

- Blackbox testing pros:
  - testing from **user perspective**
  - Can potentially **find holes** in the specification/requirements → both verification and validation
  - **Easy** to analyze and produce test cases
- Blackbox testing cons/limitations:
  - **Not implementation aware**
  - Potentially **subjective** i.e., depends on the opinions and experience of the test engineer

Putting it all together: Whitebox Testing

- Whitebox testing pros:
  - Testing from **developer perspective**
  - **Implementation-aware**
  - **More objective** i.e., not dependent on test engineer's opinion
- Whitebox testing cons/limitations:
  - **Not always available**
  - Can **miss unimplemented parts** of specs/requirements
  - Developed tests can be **more fragile** as they are tightly coupled to the specific implementation
  - **Requires high knowledge** of the code and programming in general

Putting it all together

- So, which one should we do?

(Ideally) both!

- structural testing is a *check and balance* on the specification-based tests:

****the first step of a test engineer should be to derive test cases out of any requirements-based technique. Once requirements are fully covered, test engineers then perform structural testing to cover what is missing from the structural point of view. Any divergences should be brought back to the requirements-based testing phase****

Final Note

- Blackbox and Whitebox testing are *views* we take towards testing:
  - Both can be applied at unit, integration and system levels