The coverage paradox

When 90% isn’t enough, but less might be
Initial coverage analysis

- This project has 90% coverage.
- I don’t know the details of what is tested.
- The risk associated to this project is low.
A closer inspection
Testing trivial methods:

- `init()` 100%
- `getFirstName()` 100%
- `setFirstName(String)` 100%
- `setLastName(String)` 100%
- `getLastName()` 100%
Incomplete tests:

```java
public class Calculator {
    public int divide (int numerator, int denominator) {
        return numerator / denominator;
    }
}
```

```java
public class CalculatorTest {
    @Test
    public void testDivide () {
        assertThat(new Calculator().divide(1,1)).isEqualTo(1));
    }
}
```

This class has 100% coverage.

The test only checks the behaviour of 1/1.
A look at code with no coverage:

- Written in 2010
- Last modified in 2012
- 5% of the total code
- High complexity

What is this piece of code?

- It is the Backup-Restore function...
- The most critical function in your application
- It rarely gets used
- But when it does, it has to work perfectly
Get to know your code
Metrics to know your code better

In addition to **Coverage**, other metrics can provide more information about your project:

1. Testability
2. Cyclomatic Complexity
3. Dependency analysis
4. Mutation test score
1. Testability

In order to have good tests, your code must be **testable**. A method is testable when:

- I can fully control its inputs
- I can fully observe its effects

```java
public class Counter {
    private int counter = 0;

    public void increment () {
        return counter++;
    }

    public int getCounter() {
        return this.counter;
    }
}
```

```java
public class CounterTest {
    @Test
    public void testIncrement () {
        Counter counter = new Counter();
        counter.increment();
        // the value of counter cannot be tested
    }
}
```

```java
assertThat(counter.getCounter()).isEqualTo(1);
```
2. Cyclomatic Complexity

Complexity = Edges - Nodes + 2

C = 9 - 8 + 2 = 3
2. Cyclomatic Complexity

Complexity is tightly linked to risk. It’s particularly important to test classes with high complexity to mitigate risk.

- complexity $\geq 8$
- $1 < \text{complexity} < 8$
- complexity = 1
3: Dependency analysis

**TRUE:** A class used by many others is critical.

**FALSE:** I don’t need to test classes not used by other classes.
4: Mutation test score

**Mutation tests** can check test quality by verifying their ability of catching regressions.

Mutation tests introduce regression in your codebase to verify that tests fail and don’t return false positives.
Testing small methods with little or no logic can create unnecessary noise, without adding any value to our project in terms of safety and risk prevention.

Our suggestion is to test these methods indirectly rather than explicitly writing tests for them.
Automating risk reduction with Diffblue Cover
Diffblue Cover

AI automated:

- unit test authoring for Java
- maintenance of unit tests
- highlighting risk in your code
Example test generation

```java
@ContextConfiguration(classes = {AmazonS3.class, CloudStorageService.class})
@ExtendWith(SpringExtension.class)
public class CloudStorageServiceDiffblueTest {
    @MockBean
    private AmazonS3 amazonS3;

    @Autowired
    private CloudStorageService cloudStorageService;

    @Test
    public void testUploadFileToBucket() throws SdkClientException {
        // Arrange
        PutObjectResult putObjectResult = new PutObjectResult();
        when(this.amazonS3.putObject(anyString(), anyString(), (File) any())).thenReturn(putObjectResult);
        // Act and Assert
       assertSame(putObjectResult, this.cloudStorageService.uploadFileToBucket("bucket-name", "object-key", Paths.get(System.getProperty("java.io.tmpdir"), "test.txt").toFile()));
        verify(this.amazonS3).putObject(anyString(), anyString(), (File) any());
    }
}
```
So where is the risk?
Cover is free to use in FINOS projects

Get started at diff.blue/FINOS

Drop me an email enrico.trentin@diffblue.com