CodeCoverage

```
Module Code: ELEE1149

Module Name: Software Engineering

Credits: 15

Module Leader: Seb Blair BEng(H) PGCAP MIET MIHEEM FHEA
```

Download as a PDF

What is it?

- Code coverage is a white-box testing technique.
- It verifies the extent to which developers have executed the code.
- The tools used for code coverage contain static instrumentation.
- Testers can use these to insert statement monitoring code execution at crucial points in the code.

Why perform Code Coverage?

Developers perform code coverage at the unit test level; this gives them a great vantage point to help them decide the tests they need to include. Code coverage helps them answer questions like:

- Does the unit test suite have enough tests?
- Has the code been implemented intentionally?

A higher code coverage percentage means lesser chances of overlooking unidentified bugs. It helps when you set up a minimum level that code coverage must achieve to reduce the chances of finding bugs at later stages of the development process.

Best practice

1. **Aim for Balanced Coverage Goals**: Avoid setting an arbitrary 100% coverage goal, as it can lead to unnecessary tests that may not add value. Instead, aim for practical, high-impact coverage percentages that ensure critical areas are covered.
2. **Measure Relevant Metrics**: Depending on your application requirements, focus on different coverage types, such as statement, branch, and path coverage, to ensure that all code paths have been adequately tested.
3. **Identify Gaps Early**: Use code coverage tools to identify areas with low coverage during development. This will help address them proactively and reduce the risk of issues.
4. **Encourage Developer Ownership**: Encourage developers to write unit tests for their code and track coverage as part of the development process. This increases accountability and builds a robust codebase from the start.
5. **Continuously Integrate Coverage Checks**: Integrate code coverage tools into the CI/CD pipeline to consistently monitor and enforce coverage standards with each code change.

Levels of Code coverage

Method Coverage (Function Coverage)

Testers measure code through method or function coverage by counting the number of functions a test suite calls.

```python
# Application Code
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

# Test Code
def test_add():
    assert add(2, 3) == 5

# Only `add` is tested, so function coverage = 50%.
test_add()
```

Levels of Code coverage

Statement Coverage

Statements are instructions highlighting an action a program needs to carry out. Statement coverage hence measures the percentage of code statements and gives an accurate estimate of the quantity of code the tests execute.

```python
# Application Code
def calculate(x):
    if x > 0:
        print("Positive number")
    print("Calculation done")

# Test Code
def test_calculate():
    calculate(1)  # This executes all statements in `calculate`.

# Statement coverage = 100% as all statements are executed.
test_calculate()
```

Levels of Code coverage

Branch Coverage

Branch coverage measures whether a test suite executes the branches from decision points written into the code. Such decision points arise from if and case statements, with two possible outcomes: true and false.

The goal is to verify if the tests execute all branch points across a comprehensive set of inputs. It helps testers measure code logic.

```python
# Application Code
def check_even_odd(x):
    if x % 2 == 0:
        return "Even"
    else:
        return "Odd"

# Test Code
def test_check_even_odd():
    assert check_even_odd(2) == "Even"  # Covers the `if` branch
    assert check_even_odd(3) == "Odd"  # Covers the `else` branch

# Branch coverage = 100% as both branches (`if` and `else`) are executed.
test_check_even_odd()
```

Levels of Code coverage

Condition Coverage

Developers use condition coverage to verify if tests execute statements using boolean expressions in the code - this is another way of ensuring that tests perform all possible code paths.

```python
# Application Code
def check_range(x):
    return x > 0 and x < 10

# Test Code
def test_check_range():
    assert check_range(5) == True  # True AND True
    assert check_range(-5) == False  # False AND True
    assert check_range(15) == False  # True AND False

# Condition coverage = 100% as all boolean conditions are tested.
test_check_range()
```

Levels of Code coverage

Multiple Condition Decision Coverage (MC/DC)

Each decision statement can have a combination of conditions. Multiple condition decision coverage ensures that tests execute all these combinations seamlessly. Testers use this metric to test safety-critical apps like those inside aircraft.

```python
# Application Code
def decision(a, b, c):
    return a or (b and c)

# Test Code
def test_decision():
    assert decision(True, False, False) == True  # a = True
    assert decision(False, True, True) == True  # b and c = True
    assert decision(False, False, False) == False  # All conditions False
    assert decision(False, True, False) == False  # b = True, c = False

# MC/DC ensures each condition (`a`, `b`, `c`) independently affects the outcome.
test_decision()
```

Levels of Code coverage

Parameter Value Coverage

As the name suggests, parameter coverage ensures that tests cover all possible parameter values for each program. This metric is essential as neglecting specific parameter values leads to software defects.

```python
# Application Code
def grade(score):
    if score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    else:
        return "C"

# Test Code
def test_grade():
    assert grade(95) == "A"  # score >= 90
    assert grade(85) == "B"  # 80 <= score < 90
    assert grade(70) == "C"  # score < 80

# Parameter value coverage = 100% as all possible parameter ranges are tested.
test_grade()
```

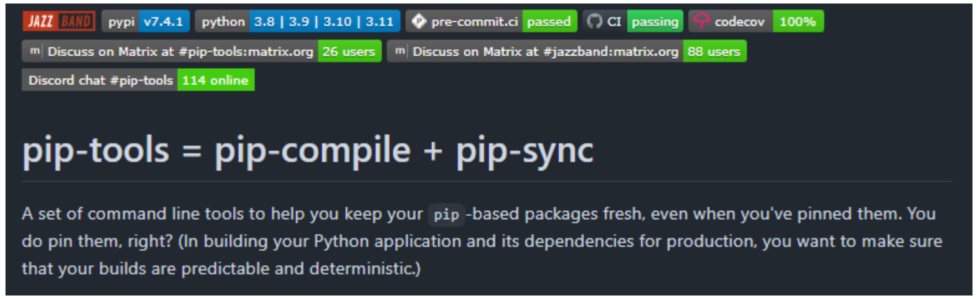
Levels of Code coverage

Cyclomatic Complexity

Testers use the cyclomatic complexity metric to measure the number of linearly dependent paths in a program's source code. It is also helpful to determine coverage for particular code modules.

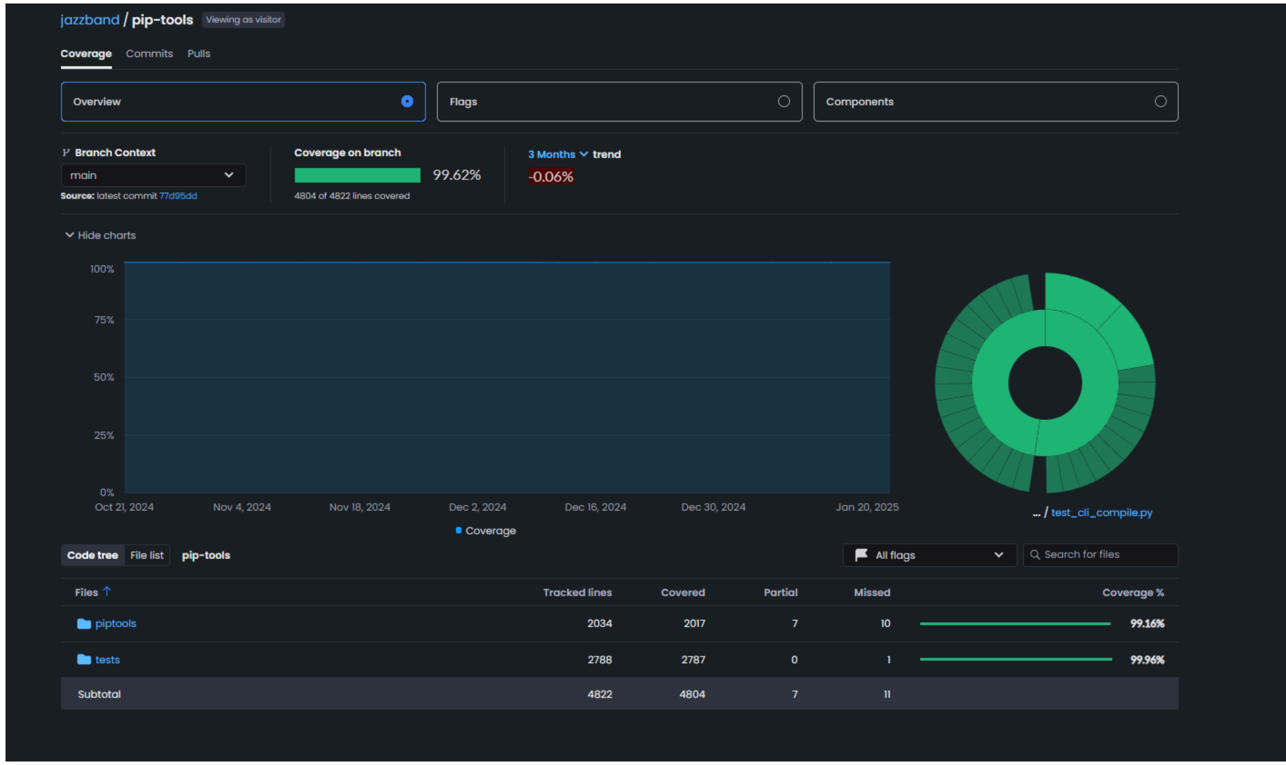
```python
# Application Code
def process(x):
    if x > 0:
        print("Positive")
    elif x == 0:
        print("Zero")
    else:
        print("Negative")

# Test Code
def test_process():
    process(1)  # Path 1: `if` branch
    process(0)  # Path 2: `elif` branch
    process(-1)  # Path 3: `else` branch

# Cyclomatic complexity = 3 (3 paths tested).
test_process()
```

Code Coverage examples


JAZZ BAND | pypi v7.4.1 | python 3.8 | 3.9 | 3.10 | 3.11 | pre-commit.ci passed | CI passing | codecov 100%
Discuss on Matrix at #pip-tools:matrix.org 26 users | Discuss on Matrix at #jazzband:matrix.org 88 users
Discord chat #pip-tools 114 online

pip-tools = pip-compile + pip-sync

A set of command line tools to help you keep your `pip`-based packages fresh, even when you've pinned them. You do pin them, right? (In building your Python application and its dependencies for production, you want to make sure that your builds are predictable and deterministic.)


jazzband / pip-tools Viewing as visitor

Coverage Commits Pulls

Overview ● | Flags ○ | Components ○

Branch Context: main
Source: latest commit 77d95dd

Coverage on branch: 99.62%
4804 of 4822 lines covered

3 Months trend: -0.06%

Files	Tracked lines	Covered	Partial	Missed	Coverage %
piptools	2034	2017	7	10	99.16%
tests	2788	2787	0	1	99.96%
Subtotal	4822	4804	7	11	