Test-driven circuit development and mutation testing

Develop a circuit against a specification and test whether your tests reject plausible defects.

Undergraduate 22 min lesson Dr. Charith Jayasekara

By the end, you'll be able to

- Write a small circuit specification before implementation.
- Build deliberate phase and order mutations.
- Interpret mutation results as evidence about a test suite.

Begin with a clear question

Test-driven development begins with a behaviour you can state precisely. For a tiny circuit, that might be "this operation is X on every input, up to global phase." Write the assertion, implement the builder and watch the assertion pass. Mutation testing then asks whether that assertion rejects intentionally damaged versions.

Build the reasoning

A mutation is a small plausible error: remove a phase gate, move it to the wrong time or use the wrong axis. A test that stays green for a meaningful wrong implementation has a gap. Some mutations are mathematically equivalent and should survive; count those separately. Mutation scores are useful only when their denominator describes which non-equivalent defects were actually considered.



Follow time from left to right. Qiskit statevector labels put q0 on the right; diagrams put q0 on the top wire.

Worked example

Use HZH as the intended X operation. Dropping Z gives HH, moving it after HH gives Z, and replacing it by X gives HXH, equivalent to Z. The reference is the independently supplied X matrix. All three non-equivalent mutations must be

rejected. This example tests the full one-qubit operation rather than only one chosen histogram.

$$U_{\text{intended}} = HZH = X, \qquad U_{\text{drop}} = HH = I, \qquad U_{\text{late}} = ZHH = Z.$$

```
from qiskit import QuantumCircuit
from qiskit.quantum_info import Operator

reference = Operator.from_label("X")
def build(mutation=None):
    c = QuantumCircuit(1)
    c.h(0)
    if mutation == "wrong-axis": c.x(0)
    elif mutation not in ["drop-z", "late-z"]: c.z(0)
    c.h(0)
    if mutation == "late-z": c.z(0)
    return c
assert Operator.from_circuit(build()).equiv(reference)
for mutation in ["drop-z", "late-z", "wrong-axis"]:
    assert not Operator.from_circuit(build(mutation)).equiv(reference)
```

Practical lab

Notebook 07 combines state, unitary, metamorphic and statistical checks with mutations. Add a reversed-control mutation and confirm that your asymmetric inputs catch it. Keep mutation generation separate from the production builder so the tests remain readable. A high score on three one-qubit defects supports that bounded claim; realistic software also needs input validation, classical post-processing, compiler-layout and execution-record tests.

Run the corresponding notebook from the software practice collection:

- Download Quantum software tests that catch plausible wrong programs

Keep a short execution record: predicted result, observed result, environment, inputs and the check that would reject a plausible defect. Explain a failure before changing the expected value.

Chapter summary

- Start from a precise observable or operation specification.
- Challenge tests with plausible non-equivalent mutations.
- Report the defects covered and improve tests around survivors.

Exercises

1. Your suite kills the three listed defects but accepts a reversed CX. What useful next test should you add?
2. Change one part of the worked program to create a plausible fault. State which assertion should fail and why.

Worked answers

1. Add asymmetric two-qubit inputs and independently specified control/target mappings, as in P14. The existing one-qubit suite provides no evidence about a CX builder.
2. A useful answer identifies a specific changed gate, parameter, mapping or model and links it to an independent expected behaviour. Keep the intended specification fixed while changing the implementation; save the failing case as a regression test.

Check your understanding

PAUSE & THINK

A mutation multiplies an isolated pure output state by a common phase. What should a physical state-equivalence test do?

- [ ] Accept it as equivalent
- [ ] Fail because every amplitude changed
- [ ] Increase shots until the common phase appears

Answer: Accept it as equivalent. A common phase preserves all probabilities for that isolated state.

Sources

IBM's bit-ordering guide, Statevector API and Operator API describe the SDK conventions and equivalence tools. The notebook bundle records its pinned environment and local execution evidence.