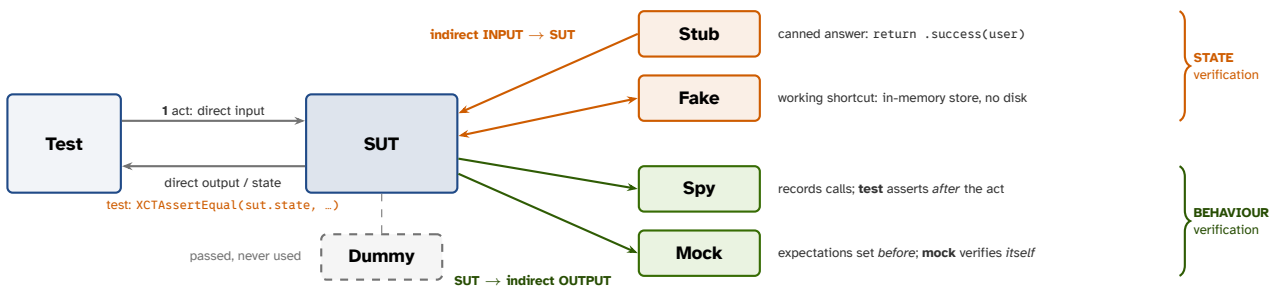


In one line: A **test double** (Gerard Meszaros, *xUnit Test Patterns*, 2007) stands in for a **DOC** so the SUT can be tested alone. **Stub/Fake** control indirect inputs → **state verification**; **Spy/Mock** observe indirect outputs → **behaviour verification**; a **Dummy** just fills a parameter. The real distinction is *direction* and *who verifies*.



How it works — Meszaros' five

- Dummy** — passed to satisfy a signature, *never used*. Make it `fatalError()` so a surprise call is loud.
- Stub** — returns *canned answers*; gives the test a *control point* for indirect inputs (success, error, empty). You then assert the SUT's state/output.
- Spy** — a stub that also *records* calls/arguments; the *test* inspects the log *after* the act. Most Swift "mocks" are really spy+stub hybrids.
- Mock** — *pre-programmed with expectations* of the calls it should receive; it *verifies itself* (fails on unexpected/missing calls).
- Fake** — a *working, simplified* implementation (in-memory repository, in-memory Core Data store). Has real logic, so it can have real bugs.
- State verification** (Stub/Fake): assert *what* the SUT ended up with — survives refactors; the default. **Behaviour verification** (Spy/Mock): assert *how* it talked to a DOC — use only when the *call is the contract* (analytics sent, logout called, email sent *once*).

Example — one protocol, five doubles

```
protocol Mailer { func send(to: String) -> Bool }

struct DummyMailer: Mailer { // fills a slot
    func send(to: String) -> Bool { fatalError("unused") } }
struct StubMailer: Mailer { // canned INPUT
    var result = false
    func send(to: String) -> Bool { result } }
final class SpyMailer: Mailer { // records OUTPUT
    private(set) var sent: [String] = []
    func send(to: String) -> Bool { sent.append(to); return true } }

final class MockMailer: Mailer { // expects + self-checks
    private let expected: [String]; private var got: [String] = []
    init(expect: [String]) { expected = expect }
    func send(to: String) -> Bool { got.append(to); return true }
    func verify(file: StaticString = #filePath, line: UInt = #line) {
        XCTAssertEqual(got, expected, file: file, line: line) } }
final class FakeMailer: Mailer { // real logic, no SMTP
    private(set) var outbox: [String] = []
    func send(to: String) -> Bool {
        guard to.contains("@") else { return false }
        outbox.append(to); return true } }
```

Remember

Dummy fills · Stub feeds · Spy notes · Mock judges · Fake works. Default: classicist (Detroit) — real objects + stubs/fakes, state verification; mockist (London) interaction checks only at true seams.

Using them — state vs behaviour

```
// STATE: the stub steers, assert the SUT's result
let sut = Signup(mailer: StubMailer(result: false))
sut.register("ada@x.io")
XCTAssertEqual(sut.state, .emailFailed)
// BEHAVIOUR: the outgoing call IS the requirement
let spy = SpyMailer()
Signup(mailer: spy).register("ada@x.io")
XCTAssertEqual(spy.sent, ["ada@x.io"]) // once, in order
```

Why iOS teams hand-roll

- Mockito (Java) builds mocks at *runtime* with proxies/bytecode; OCMock uses the Obj-C *runtime* (message forwarding) — works only for NSObject/@objc. Swift is largely *statically dispatched*, and Mirror is read-only — **no runtime mocking**.
- So Swift frameworks must **generate code**: Cuckoo, Mock-o-lo (Uber), Sourcery AutoMockable, Mockingbird, or Swift 5.9 **macros**. Cost: extra build phase, generated-file churn, breakage on Xcode/Swift upgrades, opaque code.
- A protocol double is 3–5 lines, *compiler-checked*, refactor-safe, readable in review. Lighter still: a **struct of closures** (`var fetch: (Int) async throws -> User`).

Interview traps

- Stub vs Mock** is the #1 slip: stub = *inputs*, the test asserts state; mock = *outputs*, the *double* verifies interactions.
- Over-mocking**: mocking everything (even value types) and asserting call sequences → tests pass while the app is broken, and break on every refactor. Mock only side-effecting boundaries: network, disk, clock, analytics.
- Tautology**: stub returns `User(id:1)`, assert `User(id:1)` — the decoding you care about never ran. Stub the *raw bytes*.
- Partial mock** (subclass the SUT, override some methods) — tests a different class; extract the dependency instead.
- "Don't mock what you don't own" — wrap `URLSession` behind your protocol; never subclass framework types.

Likely questions

- Stub vs mock? — canned input + state check vs expected calls, self-verified.
- Spy vs mock? — spy: test asserts after; mock: expectations before, checks itself.
- When a fake? — read-after-write / multi-call flows (in-memory repo).
- Why no Mockito in Swift? — static dispatch, no runtime proxies → codegen only.
- When behaviour verification? — when the interaction itself is the requirement.