When using JUnit 5 for testing Jenkins Pipeline Unit Sorry, Jenkins Shared Library using Jenkins Pipeline Unit with JUnit 5 to mock a class a pure groovy class or else a class which are using some groovy pipeline steps How to mock it? Do we have to use any external library? or we can use by default Jenkins Pipeline Unit or JUnit 5 provides the way to mock it or else don't give a Spock example because we are only focusing on Jenkins Pipeline Unit Is there a way?

**ChatGPT said:**

Great question 👍 You’re mixing **three things** here that behave a little differently:

1. **JUnit 5 (testing framework)** → just runs tests, gives you assertions, lifecycle (@Test, @BeforeEach etc.).  
   👉 JUnit **does not provide mocking**.
2. **Jenkins Pipeline Unit** → framework to simulate Jenkins pipeline steps and binding (sh, git, checkout, etc.) so your pipeline/shared library code can run locally.  
   👉 It lets you **register allowed methods** and **return fake results**, but it is *not* a general-purpose mocking framework.
3. **Groovy classes (plain or with pipeline steps inside)** → if you want to mock these, you need to choose between:
   * **Stubbing via Pipeline Unit** (for pipeline steps / global vars).
   * **Using a mocking library like Mockito or GroovyMock** (for plain Groovy classes).

**✅ Case 1: Mocking pipeline steps inside a shared library**

If your Groovy class or step calls something like sh, echo, etc., you handle it using **Pipeline Unit’s helper**:

// inside your test

helper.registerAllowedMethod("sh", [Map], { args ->

return 0 // fake exit code

})

helper.registerAllowedMethod("echo", [String], { msg ->

println "ECHO: ${msg}"

})

This is the "mocking" mechanism Pipeline Unit provides — you don’t need an external lib here.

**✅ Case 2: Mocking a plain Groovy class (no pipeline steps)**

For pure Groovy classes, Pipeline Unit doesn’t help — you need a **mocking library**.  
Typical choice with JUnit 5 is **Mockito**:

import org.junit.jupiter.api.Test

import org.mockito.Mockito

class MyTest {

@Test

void testWithMock() {

def dependency = Mockito.mock(MyHelperClass)

Mockito.when(dependency.someMethod()).thenReturn("fake")

def result = new MyMainClass(dependency).run()

assert result == "fake"

}

}

Here JUnit 5 just runs the test, Mockito does the mocking.

**✅ Case 3: Your class mixes pipeline steps + your own logic**

Example:

class Deploy {

def steps

def run() {

steps.sh(script: "echo hello")

return "done"

}

}

In tests, you don’t mock the Deploy class itself. Instead you:

* Give it a fake steps (provided by Pipeline Unit).
* Register allowed methods like sh.

class DeployTest extends BasePipelineTest {

@Test

void testDeploy() {

def script = loadScript("vars/deploy.groovy")

helper.registerAllowedMethod("sh", [Map], { args ->

assert args.script == "echo hello"

return 0

})

assert script.run() == "done"

}

}

**🚩 To your direct question:**

* **JUnit 5** → does NOT provide mocking.
* **Pipeline Unit** → only provides *step stubbing*, not general class mocking.
* **Plain Groovy class mocking** → you need **Mockito** (or Groovy’s built-in GroovyMock if you were using Spock, but you said no Spock).

So answer:  
👉 Use **Pipeline Unit stubs** for Jenkins steps,  
👉 Use **Mockito (with JUnit 5)** for plain Groovy class mocking.  
👉 No, Pipeline Unit alone cannot mock arbitrary Groovy classes.

Do you want me to give you a **side-by-side example** where one test uses Pipeline Unit to mock sh and another uses Mockito to mock a helper class, both under JUnit 5?

**You said:**

What if I want to use jenkins-pipeline-unit', junit 4 now what is the case

**ChatGPT said:**

Perfect, now you’ve narrowed it to **Jenkins Pipeline Unit + JUnit 4** 👍

The picture is a bit different, but the principles stay the same:

**🔹 JUnit 4 vs JUnit 5 in this context**

* **JUnit 4** is what jenkins-pipeline-unit was originally designed for → most examples in the repo/docs use @Test from JUnit 4.
* **JUnit 5** requires a JUnit Platform bridge (extra config).
* In practice → if you’re on **JUnit 4**, everything works out of the box.

**🔹 Mocking in JUnit 4 + Pipeline Unit**

**Case 1: Mocking Jenkins steps**

Handled the same way as before → via **Pipeline Unit helper**:

import com.lesfurets.jenkins.unit.BasePipelineTest

import org.junit.Before

import org.junit.Test

class DeployTest extends BasePipelineTest {

@Before

void setUp() {

super.setUp()

// Stub Jenkins steps

helper.registerAllowedMethod("sh", [Map], { args ->

println "Fake sh: ${args.script}"

return 0

})

helper.registerAllowedMethod("echo", [String], { msg ->

println "ECHO: $msg"

})

}

@Test

void testDeploy() {

def script = loadScript("vars/deploy.groovy")

def result = script.run()

assert result == "done"

}

}

👉 No external library needed — jenkins-pipeline-unit alone is enough for steps.

**Case 2: Mocking a plain Groovy class**

jenkins-pipeline-unit does **not** mock arbitrary classes.  
For that you use **Mockito (with JUnit 4)**:

import org.junit.Test

import static org.mockito.Mockito.\*

class MyServiceTest {

@Test

void testWithMock() {

def helperClass = mock(MyHelperClass)

when(helperClass.compute()).thenReturn("mocked")

def service = new MyService(helperClass)

assert service.run() == "mocked"

}

}

👉 Here JUnit 4 runs the test, Mockito handles the mocking.

**Case 3: Mixing pipeline steps + helper class**

If your Groovy code mixes both:

class Deploy {

def steps

def helper

def run() {

steps.sh(script: "echo hello")

return helper.compute()

}

}

Your test would mix **Pipeline Unit stubs** + **Mockito**:

class DeployTest extends BasePipelineTest {

@Test

void testDeploy() {

def helperClass = mock(MyHelperClass)

when(helperClass.compute()).thenReturn("mocked")

helper.registerAllowedMethod("sh", [Map], { args ->

assert args.script == "echo hello"

return 0

})

def deploy = new Deploy(steps: this, helper: helperClass)

assert deploy.run() == "mocked"

}

}

**🚩 Summary (JUnit 4 + Pipeline Unit)**

* **Pipeline steps** → mock/stub with helper.registerAllowedMethod(...) (built-in).
* **Plain Groovy classes** → mock with **Mockito** (external).
* **Combination** → mix both approaches in one test.

Do you want me to show you a **minimal Gradle build.gradle setup** for JUnit 4 + Mockito + Jenkins Pipeline Unit, so you can run these tests directly?

**You said:**

Which one is better, like by refactoring our original code to help test a unit test or using a mockito in either of JUnit 4 or JUnit 5?

**ChatGPT said:**

That’s a **classic testing trade-off question** 🔥. Let’s break it down in terms of Jenkins Shared Library + Groovy world:

**🔹 1. Refactoring code to be testable (preferred when possible)**

* If your shared library code **separates logic from Jenkins steps**, testing becomes much simpler.  
  Example:
* // good separation
* class MathHelper {
* int add(int a, int b) { a + b }
* }
* def call() {
* def result = new MathHelper().add(2, 3)
* echo "Result = $result"
* }
  + You can test MathHelper directly with **plain JUnit** (no Jenkins involved).
  + For the pipeline script, you only stub Jenkins steps (echo).

✅ **Pros**:

* Cleaner design, smaller units, easier tests.
* Fewer mocks needed → tests are stable.
* Works with just Jenkins Pipeline Unit (no external library).

🚩 **Cons**:

* Requires discipline → sometimes refactoring shared libs takes time.

**🔹 2. Using Mockito (for unrefactored code or external dependencies)**

* If your code **mixes Jenkins steps with logic or helper classes**, sometimes it’s not practical to refactor immediately.
* Mockito lets you **mock collaborators** without changing production code.

✅ **Pros**:

* Quick way to isolate behavior.
* Very flexible → can fake return values, verify interactions, throw exceptions.
* Works equally in **JUnit 4 and JUnit 5**.

🚩 **Cons**:

* Adds external dependency (Mockito).
* Over-mocking leads to brittle tests (they break when implementation changes).
* You might end up testing “how” code is written, not “what” it does.

**🔹 What’s better?**

* **Best practice** in Jenkins Shared Library testing:
  + **Refactor logic out of pipeline code** → test logic directly with JUnit.
  + **Use Pipeline Unit stubs** only for Jenkins steps (sh, git, echo, etc.).
  + **Use Mockito only when absolutely needed** (e.g., mocking a heavy helper or API client).

👉 That way:

* **80–90% of tests** → plain JUnit + Pipeline Unit (fast, stable, no external lib).
* **10–20% of tests** → Mockito for unavoidable cases.

**🔹 JUnit 4 vs JUnit 5**

* **JUnit 4** → integrates best with jenkins-pipeline-unit (mature, fewer setup headaches).
* **JUnit 5** → more modern, better extensibility, but requires a JUnit Platform runner (extra config).
* Both support Mockito equally.

👉 If you’re just starting, **JUnit 4 is simpler and battle-tested with Jenkins Shared Library**.  
If your org is already standardized on JUnit 5 → go with 5 + Mockito.

✅ **Rule of thumb**:

* Prefer **refactoring for testability** (small, pure Groovy classes).
* Use **Pipeline Unit stubs for Jenkins steps**.
* Fall back to **Mockito** when you cannot easily refactor.

Do you want me to sketch a **sample folder layout** (shared lib + tests) showing how to separate logic vs pipeline code so you see where mocking is avoided vs where it’s required?

**You said:**

so what do you suggest like do we have to refactor logic like dependency injection in our VARS folder or SRC or do we have to use Mockito for mocking please guide me here and it should not break any other future tests like performance test, sanity test, and integration test all these things those also should work not only unit testing

**ChatGPT said:**

Got it ✅ — you want **a strategy that works for unit tests, sanity tests, integration tests, performance tests, without breaking future maintainability**.

Let’s go step by step:

**🔹 Where to put logic: vars/ vs src/**

* **vars/ folder** → for pipeline *entry points* (DSL-facing). These are thin wrappers that call logic but should **not contain heavy business logic**.  
  Example (vars/deploy.groovy):
* def call(Map args = [:]) {
* def service = new com.myorg.DeployService(this) // pass pipeline context
* service.run(args)
* }
* **src/ folder** → for *logic classes*. These should be plain Groovy classes with minimal/no Jenkins step usage.  
  Example (src/com/myorg/DeployService.groovy):
* package com.myorg
* class DeployService {
* def steps // pipeline context injected
* DeployService(steps) {
* this.steps = steps
* }
* def run(Map args) {
* steps.sh(script: "echo Deploying ${args.app}")
* return "done"
* }
* }

👉 This gives you **dependency injection naturally**:

* Pipeline context (steps) is injected from vars/.
* Logic is testable in isolation (unit tests).

**🔹 Testing strategy**

1. **Unit tests (fast, isolated)**
   * Test src/ classes as much as possible.
   * Use **plain JUnit** for pure logic.
   * Use **Pipeline Unit stubs** if class calls Jenkins steps.
   * Use **Mockito** only if your class depends on another heavy/complex class.
2. **Sanity & Integration tests (Jenkins context aware)**
   * Load scripts from vars/.
   * Use Pipeline Unit to fake Jenkins environment (env, sh, git, etc.).
   * Don’t mock logic here — let it run with realistic behavior.
3. **Performance tests**
   * These are usually not unit-test-level.
   * Run your shared library against a staging Jenkins with real workloads.
   * The unit test setup won’t interfere because you’re running the real code.

**🔹 Refactor vs Mockito?**

* **Refactor to src/ with DI** → ✅ recommended. This:
  + Keeps unit tests stable (you’re testing logic, not Jenkins quirks).
  + Makes integration tests easy (only vars/ layer is pipeline-bound).
  + Reduces need for Mockito.
* **Use Mockito** → 🚨 fallback, not first choice. Use it if:
  + You depend on an external service/client you don’t want to hit in tests.
  + You cannot yet refactor a big ball of pipeline code.

**🔹 How it all fits together**

* **vars/** → thin wrappers, integration-tested.
* **src/** → logic classes, unit-tested.
* **Tests** →
  + Unit → JUnit + Pipeline Unit (rare Mockito).
  + Integration → Pipeline Unit simulating Jenkins env.
  + Sanity/Perf → full Jenkins job runs, no mocks.

This way:

* **Unit tests** stay isolated, fast.
* **Integration & sanity tests** verify correctness with Jenkins.
* **Perf tests** use real Jenkins → unaffected by mocks.

✅ **Recommendation for you:**

* Put logic in src/ with dependency injection.
* Keep vars/ thin → only wire dependencies.
* Use Pipeline Unit for Jenkins stubbing, Mockito rarely.
* This pattern will scale to unit, sanity, integration, and perf tests without conflicts.

Would you like me to draw a **sample folder structure (vars/, src/, test/)** with a concrete example so you can visualize how everything fits?

**You said:**

Usually, what it happens is like in our production grade setup, we have a VARs folder, we have a SRC folder. In SRC, usually we call SRCs from VARs logic, but internally in SRC as well, those modules actually call other utils, helper utils internally. We have a very interconnected logic in SRC as well, because let's say ABC util is calling XYZ util. In this case, what is the best way to unit test?

**ChatGPT said:**

Excellent follow-up 🙌 You’ve basically described what happens in any **production-grade Jenkins Shared Library**:

* vars/ → entry points, usually very thin.
* src/ → main services, utils, helpers, all interconnected (like micro-utils).
* Inside src/, classes call each other (e.g., ABCUtil → XYZUtil).

So the question is: **how do we test this interconnected SRC world without making tests brittle or breaking higher-level tests (sanity, perf, integration)?**

**🔹 Testing Strategy for Interconnected SRC**

Think of testing in **layers**, just like your code structure:

**1. Unit test at class level (low granularity)**

* Test **individual utils** (ABCUtil, XYZUtil) in isolation.
* If a util depends on another util:
  + Prefer **real dependency** if it’s simple (composition).
  + Use **Mockito** if the dependency is heavy, external, or expensive.

👉 Example:

class ABCUtil {

XYZUtil xyz

def compute() {

return "abc-" + xyz.process()

}

}

class ABCUtilTest {

@Test

void testCompute\_withMockedXYZ() {

def xyz = mock(XYZUtil)

when(xyz.process()).thenReturn("mocked")

def abc = new ABCUtil(xyz: xyz)

assert abc.compute() == "abc-mocked"

}

}

This way, you **isolate ABCUtil** from XYZUtil — classic unit test.

**2. Integration test at module level (medium granularity)**

* Test **a cluster of utils working together** (e.g., ABCUtil + XYZUtil) without mocks.
* Let them call each other for real — this ensures their collaboration works.

👉 Example:

class ABCXYZIntegrationTest {

@Test

void testCompute\_integration() {

def xyz = new XYZUtil()

def abc = new ABCUtil(xyz: xyz)

def result = abc.compute()

assert result == "abc-" + xyz.process() // real interaction

}

}

**3. Pipeline integration tests (high granularity)**

* Only for code in vars/.
* Use jenkins-pipeline-unit to stub Jenkins steps.
* Load script from vars/ and verify high-level flow.

👉 Example:

class DeployPipelineTest extends BasePipelineTest {

@Test

void testDeploy() {

helper.registerAllowedMethod("sh", [Map], { args -> return 0 })

def script = loadScript("vars/deploy.groovy")

script.call(app: "myApp")

printCallStack() // verify execution trace

}

}

**🔹 Rule of Thumb**

* **Inside src/**:
  + Small utils → test directly (no mocks).
  + Larger services that depend on others → use **dependency injection** + **Mockito** to isolate.
* **Between utils**:
  + Write **integration tests** to validate real wiring (no mocks).
* **In vars/**:
  + Use **Pipeline Unit stubs** to fake Jenkins steps.

**🔹 Why this works for you**

* **Unit tests** → fast, isolated, validate correctness of single classes.
* **Integration tests (src)** → ensure inter-util wiring doesn’t silently break.
* **Pipeline tests (vars)** → ensure Jenkins context wiring works.
* **Perf & sanity tests** → run against real Jenkins jobs, unaffected by mocks.

So you end up with a **test pyramid**:

Perf / Sanity (few, slow)

Pipeline integration tests (medium number)

SRC integration tests (more, medium fast)

Unit tests (lots, very fast)

✅ **Recommendation for your production setup:**

* Keep **dependency injection** in src/ (so utils can be mocked if needed).
* Write **unit tests with mocks** where isolation matters.
* Write **integration tests without mocks** to test real util chaining.
* Keep vars/ thin and integration-test them with Pipeline Unit.

Do you want me to sketch a **concrete test pyramid example with folders** (showing vars/, src/, test/unit, test/integration) so you see where each kind of test goes?

**You said:**

So in this case, do we have to have a separate Jenkins file for unit testing, integration testing, or for some other testing like sanity or something like that?

**ChatGPT said:**

Excellent question 👍 — this is where **test orchestration** comes in. You already have **different types of tests (unit, integration, sanity, perf)**, and the key is to decide **how to run them in CI/CD (Jenkins)**.

**🔹 Do we need separate Jenkinsfiles?**

**No, not necessarily.**  
In production-grade setups, we usually have **one Jenkinsfile per repo (per service/shared lib)**, but the pipeline is structured into **stages** that control which tests run.

👉 You don’t want 4 separate Jenkinsfiles (Jenkinsfile.unit, Jenkinsfile.integration, …). That gets unmanageable.  
👉 Instead, you want **one Jenkinsfile with multiple stages** or **pipeline libraries** that trigger tests selectively.

**🔹 Typical Structure**

**Jenkinsfile (Shared Library Repo Example)**

pipeline {

agent any

stages {

stage('Unit Tests') {

steps {

sh './gradlew test --tests \*UnitTest'

}

}

stage('Integration Tests') {

steps {

sh './gradlew test --tests \*IntegrationTest'

}

}

stage('Sanity Tests') {

steps {

// Sanity tests usually lightweight E2E checks

sh './gradlew test --tests \*SanityTest'

}

}

stage('Performance Tests') {

when {

branch 'main' // Only run in main or release branches

}

steps {

// Perf tests are slow, so trigger conditionally

sh './gradlew test --tests \*PerformanceTest'

}

}

}

}

**🔹 How tests are separated**

* **By naming convention** (\*UnitTest, \*IntegrationTest, …).
* Or by **Gradle source sets**:
* src/test/java → unit tests
* src/integrationTest → integration tests
* src/sanityTest → sanity tests
* src/perfTest → perf tests

Then in Jenkins, run ./gradlew unitTest, ./gradlew integrationTest, etc.

**🔹 When to run what**

* **Unit tests** → always (fast).
* **Integration tests** → on feature branches + PRs.
* **Sanity tests** → after deploy to DEV/TEST environments.
* **Performance tests** → on schedule (nightly/weekly) or before release.

**🔹 Why NOT separate Jenkinsfiles?**

* **Duplication hell** → each Jenkinsfile would repeat boilerplate.
* **Hard to maintain** → changes to the pipeline need updating multiple files.
* **Better with stages & conditions** → one Jenkinsfile, flexible control.

✅ **Recommendation for your case:**

* Stick to **a single Jenkinsfile per repo**.
* Organize tests into **stages** (unit, integration, sanity, perf).
* Use **naming convention or source sets** to separate test classes.
* Control execution with **branch conditions** (e.g., perf tests only on main).

👉 Would you like me to give you a **Gradle + Jenkinsfile setup** that separates unit, integration, sanity, and perf into different Gradle tasks, so Jenkins can call them cleanly without custom --tests filters?

**You said:**

So, give me full guide of using a Mockito along with a Jenkins Pipeline Unit using JUnit 5. All the challenges, what I need to follow, like basic things and all. It should not block me in a varying future like while mocking any pure Java, Groovy classes or a class which have a Jenkins Pipeline DSL as well. Anything and everything.

**ChatGPT said:**

Perfect 🚀 — you’re basically asking for a **playbook**:  
👉 *How do I combine Mockito + Jenkins Pipeline Unit + JUnit 5 in a way that future-proofs my testing strategy (pure Groovy/Java classes AND Jenkins DSL code)?*

Let me break it down systematically.

**🔹 1. The Core Players**

1. **JUnit 5** – test runner & assertions.
   * Gives you @Test, @BeforeEach, etc.
   * No mocking built-in.
2. **Mockito** – mocking framework.
   * Works for **pure Java classes**.
   * Works for **Groovy classes** (with some caveats: Groovy dynamic methods/final classes need extra care).
   * Provides stubbing (when(...).thenReturn(...)) and verification (verify(...)).
3. **Jenkins Pipeline Unit** – Jenkins DSL simulation.
   * Lets you run shared libraries locally.
   * Provides BasePipelineTest, loadScript(), and helper.registerAllowedMethod() for DSL stubs like sh, git, echo.

**🔹 2. Dependency Setup**

**Gradle example:**

dependencies {

testImplementation 'org.junit.jupiter:junit-jupiter-api:5.10.2'

testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.10.2'

testImplementation 'org.mockito:mockito-core:5.12.0'

testImplementation 'com.lesfurets:jenkins-pipeline-unit:1.19'

}

Enable JUnit 5 in Gradle:

test {

useJUnitPlatform()

}

**🔹 3. Test Folder Structure**

├── vars/

│ └── deploy.groovy # pipeline entry

├── src/com/myorg/

│ ├── DeployService.groovy # has Jenkins DSL + helpers

│ ├── ABCUtil.groovy # plain util

│ └── XYZUtil.groovy # plain util

└── test/groovy/com/myorg/

├── DeployServiceTest.groovy

├── ABCUtilTest.groovy

├── XYZUtilTest.groovy

└── DeployPipelineTest.groovy

**🔹 4. The Challenges & Solutions**

**Challenge 1: Mocking Jenkins DSL steps**

* Pipeline Unit does **not use Mockito** for this.
* Instead, stub them via registerAllowedMethod.

helper.registerAllowedMethod("sh", [Map], { args -> return 0 })

helper.registerAllowedMethod("echo", [String], { msg -> println "ECHO: $msg" })

👉 **Rule**: Always use **Pipeline Unit** for Jenkins DSL mocking, not Mockito.

**Challenge 2: Mocking pure Groovy/Java classes**

* Use Mockito for utils/services.
* Works fine with JUnit 5.

import org.junit.jupiter.api.Test

import static org.mockito.Mockito.\*

class ABCUtilTest {

@Test

void testCompute() {

def xyz = mock(XYZUtil)

when(xyz.process()).thenReturn("mocked")

def abc = new ABCUtil(xyz: xyz)

assert abc.compute() == "abc-mocked"

}

}

👉 **Rule**: Use Mockito for non-DSL classes.

**Challenge 3: Testing a class that mixes Jenkins DSL + utils**

class DeployService {

def steps

def helper

def run(Map args) {

steps.sh(script: "echo ${args.app}")

return helper.compute()

}

}

Test with **both** Pipeline Unit + Mockito:

class DeployServiceTest extends BasePipelineTest {

@BeforeEach

void init() { super.setUp() }

@Test

void testRun() {

helper.registerAllowedMethod("sh", [Map], { args ->

assert args.script.contains("myApp")

return 0

})

def mockHelper = mock(ABCUtil)

when(mockHelper.compute()).thenReturn("mocked")

def svc = new DeployService(steps: this, helper: mockHelper)

assert svc.run(app: "myApp") == "mocked"

verify(mockHelper).compute()

}

}

👉 **Rule**: DI (steps, helper) + Pipeline Unit for DSL + Mockito for helpers.

**Challenge 4: JUnit 5 vs JUnit 4 compatibility**

* JUnit 4 = easier with Pipeline Unit (older docs/examples).
* JUnit 5 = more modern but needs useJUnitPlatform().
* Mockito works in both.  
  👉 Choose **JUnit 5** if your org standardizes on it.

**Challenge 5: Future-proofing (sanity, perf, integration tests)**

* **Unit tests (fast)** → Mockito + Pipeline Unit.
* **Integration tests (real utils)** → Don’t mock utils; let them call each other. Still stub Jenkins DSL.
* **Sanity tests** → Run thin vars/ entry points in Pipeline Unit with minimal stubbing.
* **Perf tests** → Run on real Jenkins, no mocking.

👉 **Rule**: Don’t overuse Mockito — use it only when isolation is required.

**🔹 5. Best Practices (Playbook)**

✅ **Refactor for testability**

* Keep vars/ thin.
* Put logic in src/.
* Use DI (inject steps, inject helpers).

✅ **Mockito usage**

* For pure Groovy/Java → safe.
* Avoid mocking Jenkins DSL with Mockito (always stub via Pipeline Unit).
* Use verify() only when behavior matters (don’t over-verify).

✅ **Test layers**

* Unit: fast, isolated.
* Integration (src-to-src): no mocks.
* Pipeline: DSL stubs only.
* Perf: real Jenkins.

✅ **Naming convention**

* \*UnitTest → pure utils with Mockito.
* \*IntegrationTest → multiple utils together.
* \*PipelineTest → DSL-focused tests.

**🔹 6. Example Jenkinsfile Integration**

pipeline {

agent any

stages {

stage('Unit Tests') {

steps {

sh './gradlew test --tests \*UnitTest'

}

}

stage('Integration Tests') {

steps {

sh './gradlew test --tests \*IntegrationTest'

}

}

stage('Pipeline Tests') {

steps {

sh './gradlew test --tests \*PipelineTest'

}

}

}

}

✅ With this setup:

* You can **mock any Groovy/Java class with Mockito**.
* You can **mock Jenkins DSL only with Pipeline Unit stubs**.
* You can scale to **unit, integration, sanity, performance tests** without rewriting tests.

Do you want me to also give you a **starter Gradle project template** (with vars/, src/, test/, Gradle config, and one sample Mockito + Pipeline Unit test), so you can bootstrap this in your environment immediately?

**You said:**

Give me everything like syntax level like if someone is starting with a unit test like how to use a marketer like what exactly the syntax how to follow all these things

**ChatGPT said:**

Perfect 👍 you want **a from-scratch, syntax-level guide** that someone brand new to **unit testing with Mockito + Jenkins Pipeline Unit + JUnit 5** can follow without surprises.  
I’ll give you everything step by step: **setup → syntax → examples → patterns.**

**🔹 1. Project Setup**

**build.gradle**

plugins {

id 'groovy'

id 'java'

}

repositories {

mavenCentral()

}

dependencies {

// JUnit 5

testImplementation 'org.junit.jupiter:junit-jupiter-api:5.10.2'

testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.10.2'

// Mockito

testImplementation 'org.mockito:mockito-core:5.12.0'

// Jenkins Pipeline Unit

testImplementation 'com.lesfurets:jenkins-pipeline-unit:1.19'

}

test {

useJUnitPlatform() // Needed for JUnit 5

}

**Project layout**

jenkins-shared-lib/

├── vars/

│ └── deploy.groovy

├── src/com/myorg/

│ ├── DeployService.groovy

│ ├── ABCUtil.groovy

│ └── XYZUtil.groovy

└── test/groovy/com/myorg/

├── DeployServiceTest.groovy

├── ABCUtilTest.groovy

├── XYZUtilTest.groovy

└── DeployPipelineTest.groovy

**🔹 2. Mockito Syntax (Basics)**

**Create a mock**

def mockObj = mock(MyClass)

**Stub a method**

when(mockObj.someMethod()).thenReturn("fake value")

**Stub with arguments**

when(mockObj.compute("input")).thenReturn("output")

**Verify a method call**

verify(mockObj).someMethod()

**Verify with times**

verify(mockObj, times(2)).compute("input")

**Verify never called**

verify(mockObj, never()).dangerousMethod()

**Throw exception**

when(mockObj.doWork()).thenThrow(new RuntimeException("boom"))

**🔹 3. Jenkins Pipeline Unit Syntax**

**Register a pipeline DSL method**

helper.registerAllowedMethod("sh", [Map], { args ->

println "Fake SH: ${args.script}"

return 0

})

helper.registerAllowedMethod("echo", [String], { msg ->

println "ECHO: $msg"

})

**Load a script from vars/**

def script = loadScript("vars/deploy.groovy")

**Run it**

script.call(app: "myApp")

**Verify pipeline calls**

printCallStack() // prints what steps were called

**🔹 4. Example: Pure Groovy Util (Unit Test with Mockito)**

src/com/myorg/ABCUtil.groovy

package com.myorg

class ABCUtil {

XYZUtil xyz

String compute() {

return "abc-" + xyz.process()

}

}

src/com/myorg/XYZUtil.groovy

package com.myorg

class XYZUtil {

String process() {

return "real"

}

}

test/groovy/com/myorg/ABCUtilTest.groovy

package com.myorg

import org.junit.jupiter.api.Test

import static org.mockito.Mockito.\*

class ABCUtilTest {

@Test

void testCompute\_withMockedXYZ() {

def xyz = mock(XYZUtil)

when(xyz.process()).thenReturn("mocked")

def abc = new ABCUtil(xyz: xyz)

assert abc.compute() == "abc-mocked"

verify(xyz).process()

}

}

**🔹 5. Example: Class with Jenkins DSL + Helpers**

src/com/myorg/DeployService.groovy

package com.myorg

class DeployService {

def steps

def helper

String run(Map args) {

steps.sh(script: "echo Deploying ${args.app}")

return helper.compute()

}

}

test/groovy/com/myorg/DeployServiceTest.groovy

package com.myorg

import com.lesfurets.jenkins.unit.BasePipelineTest

import org.junit.jupiter.api.BeforeEach

import org.junit.jupiter.api.Test

import static org.mockito.Mockito.\*

class DeployServiceTest extends BasePipelineTest {

@BeforeEach

void setup() {

super.setUp()

// Mock Jenkins DSL step

helper.registerAllowedMethod("sh", [Map], { args ->

assert args.script.contains("myApp")

return 0

})

}

@Test

void testRun\_withMockHelper() {

def helperClass = mock(ABCUtil)

when(helperClass.compute()).thenReturn("mocked")

def service = new DeployService(steps: this, helper: helperClass)

assert service.run(app: "myApp") == "mocked"

verify(helperClass).compute()

}

}

**🔹 6. Example: Pipeline Script in vars/**

vars/deploy.groovy

def call(Map args = [:]) {

def service = new com.myorg.DeployService(this, new com.myorg.ABCUtil(xyz: new com.myorg.XYZUtil()))

return service.run(args)

}

test/groovy/com/myorg/DeployPipelineTest.groovy

package com.myorg

import com.lesfurets.jenkins.unit.BasePipelineTest

import org.junit.jupiter.api.BeforeEach

import org.junit.jupiter.api.Test

class DeployPipelineTest extends BasePipelineTest {

@BeforeEach

void setup() {

super.setUp()

helper.registerAllowedMethod("sh", [Map], { args ->

println "SH CALLED: ${args.script}"

return 0

})

}

@Test

void testPipelineScript() {

def script = loadScript("vars/deploy.groovy")

def result = script.call(app: "myApp")

assert result.startsWith("abc-")

printCallStack()

}

}

**🔹 7. Best Practices Recap**

* **Use Mockito** for:
  + Pure Groovy/Java classes in src/.
  + Isolating dependencies when needed.
* **Use Pipeline Unit** for:
  + Mocking Jenkins steps (sh, echo, git, …).
  + Testing vars/ entry points.
* **Structure**:
  + vars/ → thin, integration-tested.
  + src/ → heavy logic, unit-tested (Mockito).
  + test/ → split unit vs pipeline vs integration.
* **Verify only behavior that matters** → don’t overuse verify().
* **Refactor for dependency injection** so Mockito + Pipeline Unit can work together.

✅ With this playbook, you can:

* Mock *any* class with Mockito (Groovy or Java).
* Mock *any* Jenkins DSL step with Pipeline Unit.
* Write **unit, integration, and pipeline tests** without conflicts.
* Future-proof for sanity/perf tests (which run real code, not mocks).

Would you like me to also give you a **ready-to-clone GitHub repo template** (with Gradle, JUnit 5, Mockito, Pipeline Unit, and example tests) so you can bootstrap immediately instead of writing boilerplate?

**You said:**

So, give me a checklist kind of thing, or else a tabular kind of thing, where I should go for Jenkins pipeline unit mocking, or refactoring the original code, or this, mock it all, in JUnit file.

**ChatGPT said:**

Perfect question 👍 — this is exactly the kind of decision-making DevOps teams struggle with when testing **Jenkins Shared Libraries**. Let me give you a **decision checklist/table** that acts like a rulebook for when to use **Jenkins Pipeline Unit (JPU) DSL mocking**, when to **refactor (DI, helper extraction)**, and when to use **Mockito** inside your JUnit 5 tests.

**✅ Jenkins Shared Library Testing Decision Guide**

| **Case** | **Example** | **Best Approach** | **Why** |
| --- | --- | --- | --- |
| **Pipeline DSL calls** (e.g., sh, checkout, withCredentials) | sh("mvn clean install") inside vars/build.groovy | **Jenkins Pipeline Unit DSL stubbing** (helper.registerAllowedMethod) | JPU is designed to simulate Jenkins core steps → No need for Mockito. Keep mocking at DSL level. |
| **Vars calling other Vars** | deployApp.groovy calling buildApp.groovy | **Refactor into src/ and inject dependencies**, OR register via loadPipelineScriptForTest | Prevents spaghetti coupling. Easier to test each vars/ separately. |
| **Pure Groovy/Java utils in src/ (no DSL)** | StringUtils.capitalize(), MathUtils.calculate() | **JUnit + Mockito** | These are normal code units → test them like any other library. Mockito can mock collaborators. |
| **Groovy/Java utils in src/ that internally call Jenkins DSL** | GitUtils.cloneRepo() internally calls checkout | **Refactor to separate DSL layer** → extract DSL calls into adapter/facade; mock facade with Mockito | Mixing utils with Jenkins DSL makes unit testing fragile. Separation of concerns helps. |
| **Complex helper classes with multiple cross-dependencies** | ABCUtils → calls XYZUtils → calls LMNUtils | **Use Mockito to mock deep dependencies** in unit test | Don’t test entire dependency tree at once. Each util should be testable independently. |
| **Integration-like tests (ensure Vars + Src + DSL work together)** | Running whole vars/deploy.groovy with real DSL mock responses | **Use Jenkins Pipeline Unit only** | Treat as "integration test" at Jenkins level. Do not use Mockito here. |
| **Sanity/Smoke/Performance tests** | Running real pipelines in DEV cluster | **Do not mock at all** | These are system-level → run against real Jenkins + agents, not unit-level. |

**🔑 General Rules**

1. **Always prefer DSL stubs for pipeline steps.**
   * If it’s a Jenkins step, stub it with JPU, not Mockito.
   * Example:
   * helper.registerAllowedMethod("sh", [Map.class], { args -> return 0 })
2. **Use Mockito for business logic (non-DSL).**
   * Example:
   * @Mock GitUtils gitUtils;
   * when(gitUtils.cloneRepo(anyString())).thenReturn("success");
3. **Refactor when DSL and logic are tightly coupled.**
   * Extract DSL → Facade/Adapter layer (JPU covers it).
   * Keep utils (Groovy/Java) pure so Mockito can handle them.
4. **One test framework = one purpose:**
   * **JPU → Jenkins DSL behavior**
   * **Mockito + JUnit → Utils/business logic**
   * **Integration → DSL + Vars working together**
   * **System tests → Real Jenkins pipelines**
5. **Keep CI pipelines flexible**
   * Don’t duplicate Jenkinsfiles per test type.
   * Instead:
     + Unit tests = Gradle/Maven test phase
     + Integration tests = separate Gradle task (integrationTest)
     + Sanity/Perf = separate Jenkins pipeline jobs

**🚦 Checklist Before Writing a Test**

* Does the code call a Jenkins step (DSL)?  
  → Use JPU stub.
* Does the code call another util/helper?  
  → Mock with Mockito if you don’t want real execution.
* Is it too tightly coupled (DSL + utils)?  
  → Refactor into layers:
  + **Facade (DSL)** → JPU tested
  + **Utils** → Mockito tested
* Do I want end-to-end validation?  
  → Use JPU only, no Mockito.

👉 In short:

* **DSL = JPU**
* **Business logic = Mockito**
* **Mixed DSL+Logic = Refactor**

Would you like me to now **show you side-by-side syntax examples** of the **same function tested in three ways** → (1) JPU DSL stub, (2) Mockito, and (3) Refactor+Mockito? That’ll make this checklist super concrete.

**You said:**

So, give me a checklist kind of thing, or else a tabular kind of thing, where I should go for Jenkins pipeline unit mocking, or refactoring the original code, or this, mock it all, in JUnit file. I need to update as a documentation please

**ChatGPT said:**

Got it ✅  
I’ll prepare a **documentation-style guide** (tabular checklist) that you can plug into your repo wiki or team docs. This will show **when to use Jenkins Pipeline Unit mocking, when to refactor code, and when to use Mockito inside JUnit (4/5)** so your tests stay maintainable and production-grade.

**📘 Testing Strategy for Jenkins Shared Libraries**

| **Case** | **Preferred Approach** | **How to Do It** | **Why / Notes** |
| --- | --- | --- | --- |
| **Pipeline DSL (e.g. sh, echo, checkout)** | Jenkins Pipeline Unit mocking | Use helper.registerAllowedMethod("sh", [Map]) { ... } | Jenkins Pipeline Unit is built to mock DSL. No need for external libs. Keep it lightweight. |
| **Pure Groovy/Java utility classes (no DSL inside)** | Mockito (JUnit 4/5) | @Mock, Mockito.when(...).thenReturn(...) | Easier and faster than wiring with Pipeline Unit. Keeps test isolated. |
| **Groovy classes that internally call Pipeline DSL** | **Refactor with Dependency Injection + Jenkins Pipeline Unit** | Move DSL calls to a wrapper/helper; inject helper into your class. Test logic with Mockito; test DSL with Pipeline Unit. | Keeps DSL isolated, avoids heavy mocking of Jenkins steps. Improves maintainability. |
| **SRC classes calling other SRC utils (e.g., ABCUtil → XYZUtil)** | Mockito (JUnit 4/5) OR Refactor if dependency is DSL-heavy | Use @InjectMocks + @Mock for sub-dependencies | Avoid testing downstream utils in every test → focus only on class under test. |
| **VARS scripts (entry points for pipelines)** | Jenkins Pipeline Unit mocking + selective refactoring | Mock DSL with Pipeline Unit. If calling SRC utils, mock them with Mockito. | VARS are closest to Jenkinsfile, so they need both worlds. |
| **Performance, Sanity, Integration tests** | Do NOT mock → run real pipeline in a test/staging Jenkins | Only use minimal stubs (if needed) | Unit test is for fast feedback. System-level tests should run end-to-end. Keep separation. |
| **Shared Library “global vars” that do orchestration** | Refactor into smaller SRC utils, then unit test utils with Mockito | Example: split “deployment.groovy” into K8sUtils + AwsUtils, test each with Mockito | Avoid giant mocks; smaller, composable modules are easier to test and reuse. |
| **Complex pipelines with nested utils** | Hybrid (Refactor + Mockito + Pipeline Unit) | Refactor logic into utils → use Mockito. Keep DSL calls mocked in Pipeline Unit. | Scales well for large interconnected libraries. |

**🛠 Practical Rules of Thumb**

1. **Default rule**:
   * DSL → Jenkins Pipeline Unit
   * Plain logic → Mockito
   * DSL + logic mixed → Refactor
2. **Keep tests fast**: unit tests should **not run shell commands, Docker, AWS, etc.**. Always mock.
3. **Test layers separately**:
   * **Unit test (Mockito + JPU)** → runs in seconds
   * **Integration test** → runs in Jenkins test instance, uses real plugins/resources
   * **E2E / Sanity test** → deploys real pipeline to staging
4. **Dependency Injection pays off**: If your SRC classes are highly interconnected, refactor so each class depends on small injected helpers. This makes mocking with Mockito simple.
5. **Future-proofing**:
   * Write **unit tests** using JUnit + Mockito for reusable logic.
   * Write **pipeline tests** using Jenkins Pipeline Unit for DSL.
   * Don’t couple tests to implementation details of Jenkinsfile; test *behaviors*.

✅ With this doc, your team can **decide systematically**:

* Use Jenkins Pipeline Unit when DSL is involved.
* Use Mockito for internal logic.
* Refactor when mixing logic + DSL.