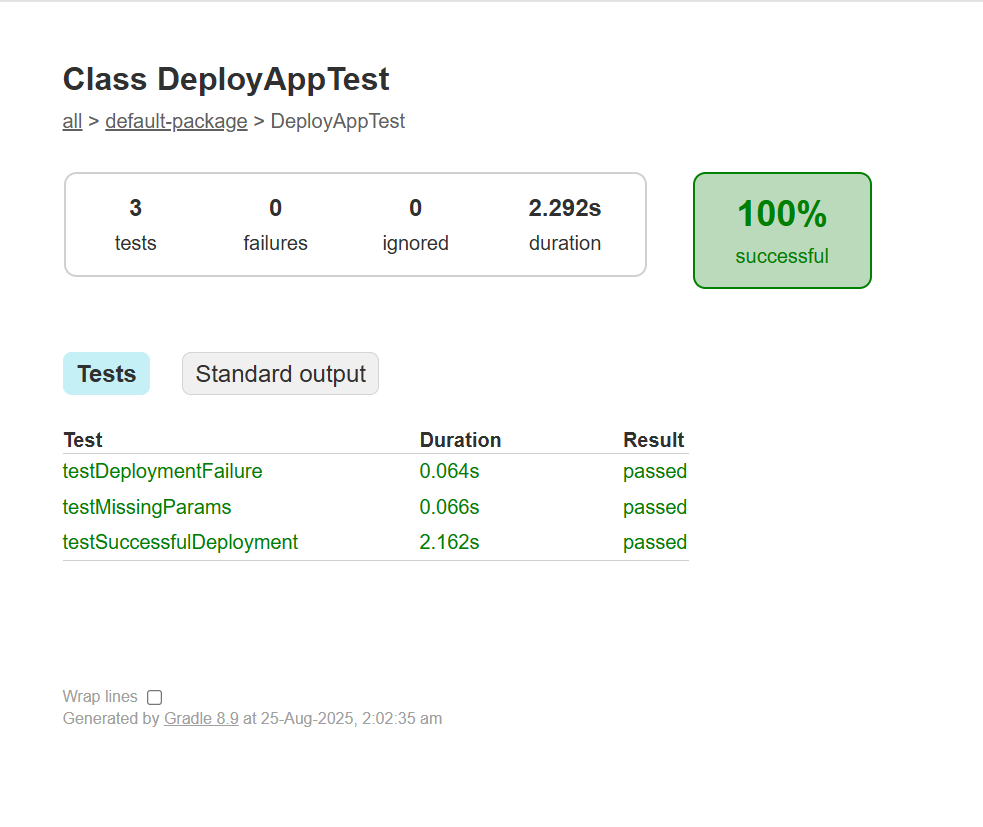
Full repo with build reports.

[jenkins-shared-lib](https://1drv.ms/f/c/4fbb2235624125eb/Ej8bMhhTwahAhCrX5U80xLYBim1FQ6ixGFUJXH6895TPCQ?e=r4IkHH)



The snippet you showed:

def mockHelper = Mock(JenkinsHelper)

def deployer = new Deployer(mockHelper)

deployer.deploy()

👉 This is **not JUnit (JPU)**.  
 It comes from **Spock Framework** — a testing framework for Groovy/Java.

**Why it’s Spock:**

* The Mock() syntax is specific to **Spock**.

In JUnit (classic with Mockito), you’d see something like:  
  
 JenkinsHelper mockHelper = Mockito.mock(JenkinsHelper.class);

 or in Groovy with Mockito:  
  
 def mockHelper = Mock(JenkinsHelper)  // only works in Spock

* Spock integrates mocking/stubbing directly with its DSL (Mock(), Stub(), Spy()), no need for external libs like Mockito in most cases.

✅ Typical Spock test:

import spock.lang.Specification

class DeployerSpec extends Specification {

    def "should call deploy on JenkinsHelper"() {

        given:

        def mockHelper = Mock(JenkinsHelper)

        def deployer = new Deployer(mockHelper)

        when:

        deployer.deploy()

        then:

        1 \* mockHelper.deploy()   // verifies that deploy() was called exactly once

    }

}

⚡ Summary:

* Mock() → **Spock framework** feature.
* JUnit alone does not provide mocking — you’d need **Mockito** or another library.
* In Groovy/Spock, mocks are **built-in**, which is why you see that syntax.

Do you want me to also show you the **equivalent JUnit + Mockito version** of your snippet for comparison?

**You said:**

Can;t we do mocking using jenkinspipelineunit

**ChatGPT said:**

Great question 👌 — let’s clear this up.

**🔹 1. JenkinsPipelineUnit**

* **JenkinsPipelineUnit** is a Groovy testing framework specifically for **Jenkins Shared Libraries** and **Declarative/Scripted Pipelines**.
* It gives you a fake Jenkins runtime where you can:  
    
  + Load vars/ steps or src/ classes,
  + Register mocks for pipeline steps (sh, git, echo, etc.),
  + Verify interactions (assertJobStatusSuccess(), assertCallStackContains()).

➡️ But it’s **not a general-purpose mocking framework** like Spock or Mockito.  
 It mocks **Jenkins pipeline steps**, not your own classes (unless you wire them manually).

**🔹 2. Spock / Mockito**

* These are **general testing frameworks**:  
    
  + **Spock** (Groovy-native) → Mock(), Stub(), Spy().
  + **Mockito** (Java/Groovy) → Mockito.mock(Class).

➡️ They let you mock **your classes** (like JenkinsHelper, Deployer), independent of Jenkins.

**🔹 3. So in your case:**

def mockHelper = Mock(JenkinsHelper)

def deployer = new Deployer(mockHelper)

deployer.deploy()

This is **Spock-style mocking**.  
 You **can’t** do this directly with JenkinsPipelineUnit, because it doesn’t provide Mock().

With JenkinsPipelineUnit, you’d instead do:

helper.registerAllowedMethod("deploy", [String]) { arg ->

    return "fake deploy for ${arg}"

}

That mocks a **pipeline step** called deploy, not your JenkinsHelper class.

✅ **Rule of Thumb**:

* Use **JenkinsPipelineUnit** → when testing Jenkins vars/ steps and pipelines.
* Use **Spock/Mockito** → when testing plain Groovy classes (e.g., JenkinsHelper, Deployer).
* You can **combine them** if your shared library mixes pipeline steps (vars/) and helper classes (src/).

[Skip to content](https://chatgpt.com/c/68ac0d53-2c14-8320-886f-b8c532f39337#main)

**Chat history**

**You said:**

Can we have a simple var folder where we can write a unit test for that one using jenkinspipelineunit using gradle 8.13 and java 21

**ChatGPT said:**

Got it 👍 You want a simple vars/ folder structure for a Jenkins Shared Library and a unit test for one of the steps using JenkinsPipelineUnit, with Gradle 8.1.3 and Java 21.

Here’s a minimal working setup:

**📂 Project Structure**

jenkins-shared-lib/

├── build.gradle

├── settings.gradle

├── src

│   └── test

│       └── groovy

│           └── MyStepTest.groovy

└── vars

    └── myStep.groovy

**⚙️** build.gradle

plugins { id 'groovy' id 'java' } group = 'com.example' version = '1.0.0' repositories { mavenCentral() } dependencies { implementation localGroovy() // JenkinsPipelineUnit testImplementation 'com.lesfurets:jenkins-pipeline-unit:1.16' // JUnit 4 testImplementation 'junit:junit:4.13.2' // Optional assertions testImplementation 'org.assertj:assertj-core:3.25.3' }

**⚙️** settings.gradle

rootProject.name = 'jenkins-shared-lib'

**📂** vars/myStep.groovy

This is the Shared Library step we’ll test.

def call(String name = 'World') { echo "Hello, ${name}!" }

**📂** src/test/groovy/MyStepTest.groovy

Unit test using JenkinsPipelineUnit.

import com.lesfurets.jenkins.unit.BasePipelineTest import org.junit.Before import org.junit.Test class MyStepTest extends BasePipelineTest { @Before void setUp() { super.setUp() // load the script from vars helper.registerAllowedMethod("echo", [String.class], { s -> println "ECHO: $s" }) } @Test void testDefaultHello() { def script = loadScript("vars/myStep.groovy") script.call() printCallStack() assertTrue helper.callStack.find { call -> call.methodName == "echo" && callArgsToString(call).contains("Hello, World!") } != null } @Test void testCustomHello() { def script = loadScript("vars/myStep.groovy") script.call("Sumanth") printCallStack() assertTrue helper.callStack.find { call -> call.methodName == "echo" && callArgsToString(call).contains("Hello, Sumanth!") } != null } }

**▶️ Running the tests**

./gradlew clean test

✅ This setup will:

* Use Gradle 8.1.3 + Java 21
* Load a step from vars/
* Mock echo
* Assert that it was called correctly

Would you like me to also extend this with a realistic example step (e.g., running sh or reading environment variables) so you can see how JenkinsPipelineUnit handles more complex cases?

**You said:**

Give me with out mocking can we run

**ChatGPT said:**

Good question 👍

Short answer: No, you can’t run Jenkins vars/ steps without mocking — at least not fully.  
Here’s why:

**🚨 Why mocking is needed**

* Jenkins steps (echo, sh, checkout, etc.) only exist inside a Jenkins runtime.
* When you run unit tests locally (Gradle + JenkinsPipelineUnit), you’re outside Jenkins.
* JenkinsPipelineUnit provides a “fake Jenkins runtime” by mocking those steps so your script doesn’t crash.

If you don’t mock, you’ll see errors like:

groovy.lang.MissingMethodException: No signature of method: myStep.echo() is applicable...

**✅ What you can do without mocking**

If your vars/ function is pure Groovy logic (no Jenkins DSL steps like echo, sh, git), then you can run tests without mocking. Example:

vars/addNumbers.groovy

def call(int a, int b) { return a + b }

src/test/groovy/AddNumbersTest.groovy

import com.lesfurets.jenkins.unit.BasePipelineTest import org.junit.Before import org.junit.Test import static org.junit.Assert.\* class AddNumbersTest extends BasePipelineTest { @Before void setUp() { super.setUp() } @Test void testAddition() { def script = loadScript("vars/addNumbers.groovy") def result = script.call(3, 7) assertEquals(10, result) } }

Run with:

./gradlew test

👉 This works fine without mocking because it’s just Groovy code, no Jenkins steps.

**🔑 Rule of Thumb**

* Uses only Groovy logic → no mocks needed ✅
* Uses Jenkins steps (echo, sh, etc.) → must mock ❌

Would you like me to show you a hybrid approach — where the step has Jenkins DSL calls but you isolate the core logic into a utility (so you can test the logic without mocks)?

**You said:**

what if i have to use the pure groovy class or vars file still not mock but unit test it using jenkinspipelineunit

**ChatGPT said:**

Nice follow-up 👌  
You’re asking: *“If I write a vars/ step or a Groovy class, can I unit test it with JenkinsPipelineUnit without mocking anything?”*

**✅ Case 1: Pure Groovy logic inside a** vars/ **file**

Yes, you can. If your vars/ file doesn’t use Jenkins pipeline steps (echo, sh, checkout, etc.), then JenkinsPipelineUnit is basically just a loader. You can loadScript and call your method directly.

**Example** vars/myMath.groovy

def call(int a, int b) { return a \* b } def square(int x) { return x \* x }

**Example** src/test/groovy/MyMathTest.groovy

import com.lesfurets.jenkins.unit.BasePipelineTest import org.junit.Before import org.junit.Test import static org.junit.Assert.\* class MyMathTest extends BasePipelineTest { @Before void setUp() { super.setUp() } @Test void testMultiplication() { def script = loadScript("vars/myMath.groovy") def result = script.call(3, 4) assertEquals(12, result) } @Test void testSquare() { def script = loadScript("vars/myMath.groovy") def result = script.square(5) assertEquals(25, result) } }

👉 Here we’re not mocking anything. The test just runs the Groovy logic.

**✅ Case 2: Pure Groovy Class (under** src/**)**

If you put your reusable logic in src/main/groovy, you don’t even need JenkinsPipelineUnit.  
You can just write a normal JUnit test.

**Example** src/main/groovy/com/example/MathUtils.groovy

package com.example class MathUtils { static int add(int a, int b) { return a + b } }

**Example** src/test/groovy/com/example/MathUtilsTest.groovy

import org.junit.Test import static org.junit.Assert.\* class MathUtilsTest { @Test void testAdd() { assertEquals(8, com.example.MathUtils.add(3, 5)) } }

👉 No mocks. No JenkinsPipelineUnit. Just plain JUnit.

**🚨 When JenkinsPipelineUnit really matters**

You only need JenkinsPipelineUnit when:

* The script uses Jenkins pipeline steps (echo, sh, git, checkout, parallel, etc.)
* You want to simulate Jenkins behavior locally

Otherwise, for pure Groovy (logic-only vars/ or classes), it’s simpler to just run normal JUnit tests.

⚡ Recommendation:  
For production-grade shared libraries →

* Put business logic in classes (src/main/groovy) → test with plain JUnit (no mocks).
* Keep vars/ steps as thin wrappers that just call those classes → only mock Jenkins steps if needed.

👉 Do you want me to show you an example where vars/ uses Jenkins steps (sh, echo) but logic is in a pure Groovy class, so the class can be tested without mocks, and only the wrapper needs JenkinsPipelineUnit?

**You said:**

Yes give me production level example where it uses vars/  and also if required src/ with and without mocking

**ChatGPT said:**

Perfect 💯 — this is the production-grade pattern most teams use for Jenkins Shared Libraries.

We’ll build an example with 3 layers:

1. vars/ step (thin wrapper) → interacts with Jenkins DSL (sh, echo, git, etc.)
2. src/main/groovy utility class → holds business logic (pure Groovy, testable without mocks)
3. Unit tests:
   * JUnit tests for the utility (no mocks needed ✅)
   * JenkinsPipelineUnit tests for the vars/ step (mocks needed ❌)

**📂 Project Structure**

jenkins-shared-lib/

├── build.gradle

├── settings.gradle

├── vars

│   └── deployApp.groovy

├── src

│   └── main

│       └── groovy

│           └── com/example/DeployUtils.groovy

└── src

    └── test

        └── groovy

            ├── DeployUtilsTest.groovy

            └── DeployAppStepTest.groovy

**⚙️** build.gradle

(same as before, Gradle 8.1.3 + Java 21 compatible)

plugins { id 'groovy' id 'java' } group = 'com.example' version = '1.0.0' repositories { mavenCentral() } dependencies { implementation localGroovy() // JenkinsPipelineUnit for vars testing testImplementation 'com.lesfurets:jenkins-pipeline-unit:1.16' // JUnit testImplementation 'junit:junit:4.13.2' // Optional assertions testImplementation 'org.assertj:assertj-core:3.25.3' }

**📂** vars/deployApp.groovy

Wrapper step that Jenkinsfiles can call.

import com.example.DeployUtils def call(String env) { echo "Starting deployment to ${env}" // Business logic: determine the server for this env def server = DeployUtils.getServerForEnv(env) // Run shell command for deployment sh "echo Deploying application to ${server}" echo "Deployment finished for ${env}" }

**📂** src/main/groovy/com/example/DeployUtils.groovy

Business logic, pure Groovy.

package com.example class DeployUtils { static String getServerForEnv(String env) { switch(env?.toLowerCase()) { case "dev": return "dev.mycompany.com" case "test": return "test.mycompany.com" case "prod": return "prod.mycompany.com" default: throw new IllegalArgumentException("Unknown environment: $env") } } }

**📂** src/test/groovy/DeployUtilsTest.groovy

✅ Pure JUnit test (no mocking).

import org.junit.Test import static org.junit.Assert.\* class DeployUtilsTest { @Test void testValidEnvironments() { assertEquals("dev.mycompany.com", com.example.DeployUtils.getServerForEnv("dev")) assertEquals("test.mycompany.com", com.example.DeployUtils.getServerForEnv("test")) assertEquals("prod.mycompany.com", com.example.DeployUtils.getServerForEnv("prod")) } @Test(expected = IllegalArgumentException) void testInvalidEnvironment() { com.example.DeployUtils.getServerForEnv("stage") } }

**📂** src/test/groovy/DeployAppStepTest.groovy

✅ JenkinsPipelineUnit test (mocking Jenkins steps).

import com.lesfurets.jenkins.unit.BasePipelineTest import org.junit.Before import org.junit.Test import static org.junit.Assert.\* class DeployAppStepTest extends BasePipelineTest { @Before void setUp() { super.setUp() // Register Jenkins pipeline DSL methods we expect helper.registerAllowedMethod("echo", [String.class], { s -> println "ECHO: $s" }) helper.registerAllowedMethod("sh", [String.class], { s -> println "SH: $s" }) } @Test void testDeployToDev() { def script = loadScript("vars/deployApp.groovy") script.call("dev") printCallStack() // debug output assertTrue helper.callStack.find { call -> call.methodName == "sh" && callArgsToString(call).contains("dev.mycompany.com") } != null } @Test void testDeployToProd() { def script = loadScript("vars/deployApp.groovy") script.call("prod") assertTrue helper.callStack.find { call -> call.methodName == "sh" && callArgsToString(call).contains("prod.mycompany.com") } != null } }

**▶️ Run tests**

./gradlew clean test

**🔑 Key Takeaways**

* vars/: Keep it thin. Handles Jenkins DSL (echo, sh, etc.). Needs JenkinsPipelineUnit + mocks for testing.
* src/: Holds reusable pure Groovy classes (business logic). Can be tested with plain JUnit (no mocks needed).
* This separation = production best practice ✅ → logic is easy to test, Jenkins-specific bits are isolated.

Do you want me to also show you how to wire this library into a Jenkinsfile (so you can actually call deployApp("dev") in a pipeline), or just focus on unit testing part?

**You said:**

give some other simple example

**ChatGPT said:**

Got it 👍 Let’s do a different, simpler production-style example that still shows both sides (vars/ wrapper + src/ pure Groovy + tests).

**📂 Project Structure**

jenkins-shared-lib/

├── build.gradle

├── vars

│   └── greetUser.groovy

├── src

│   └── main

│       └── groovy

│           └── com/example/GreetUtils.groovy

└── src

    └── test

        └── groovy

            ├── GreetUtilsTest.groovy

            └── GreetUserStepTest.groovy

**⚙️** vars/greetUser.groovy

Thin wrapper around Jenkins DSL.

import com.example.GreetUtils def call(String name) { echo "Preparing to greet ${name}" def greeting = GreetUtils.buildGreeting(name) echo greeting return greeting }

**📂** src/main/groovy/com/example/GreetUtils.groovy

Reusable logic — no Jenkins DSL here.

package com.example class GreetUtils { static String buildGreeting(String name) { if (!name?.trim()) { throw new IllegalArgumentException("Name must not be empty") } return "Hello, ${name.capitalize()}!" } }

**📂** src/test/groovy/GreetUtilsTest.groovy

Plain JUnit test → no mocks needed.

import org.junit.Test import static org.junit.Assert.\* class GreetUtilsTest { @Test void testGreeting() { assertEquals("Hello, Sumanth!", com.example.GreetUtils.buildGreeting("sumanth")) } @Test(expected = IllegalArgumentException) void testEmptyNameFails() { com.example.GreetUtils.buildGreeting("") } }

**📂** src/test/groovy/GreetUserStepTest.groovy

PipelineUnit test for the vars/ wrapper → mock Jenkins steps.

import com.lesfurets.jenkins.unit.BasePipelineTest import org.junit.Before import org.junit.Test import static org.junit.Assert.\* class GreetUserStepTest extends BasePipelineTest { @Before void setUp() { super.setUp() helper.registerAllowedMethod("echo", [String.class], { s -> println "ECHO: $s" }) } @Test void testGreetUserStep() { def script = loadScript("vars/greetUser.groovy") def result = script.call("sumanth") printCallStack() assertEquals("Hello, Sumanth!", result) assertTrue helper.callStack.find { call -> call.methodName == "echo" && callArgsToString(call).contains("Hello, Sumanth!") } != null } }

**▶️ Run it**

./gradlew clean test

✅ This is simpler than the deployment example, but still production-ready because:

* vars/greetUser.groovy → Jenkins DSL (needs mocking).
* src/com/example/GreetUtils.groovy → pure Groovy (no mocks, plain JUnit).
* Tests are split accordingly.

👉 Do you want me to also give you a Jenkinsfile snippet that uses this shared lib (greetUser("sumanth")) so you can test it end-to-end on Jenkins?

**You said:**

what if we have pure var folder how we can unit test with mock and without mock

**ChatGPT said:**

Great question 🚀 — let’s now focus only on the vars/ folder (no src/ classes).

We’ll look at two cases:

1. vars/ uses only Groovy logic → test without mocks
2. vars/ uses Jenkins pipeline steps (like echo, sh) → test with mocks

**📂 Project: Pure** vars/ **Folder**

jenkins-shared-lib/

├── build.gradle

├── vars

│   ├── mathStep.groovy

│   └── echoStep.groovy

└── src

    └── test

        └── groovy

            ├── MathStepTest.groovy

            └── EchoStepTest.groovy

**⚙️** build.gradle

(same as before, works with Gradle 8.1.3 + Java 21)

plugins { id 'groovy' id 'java' } group = 'com.example' version = '1.0.0' repositories { mavenCentral() } dependencies { implementation localGroovy() // JenkinsPipelineUnit testImplementation 'com.lesfurets:jenkins-pipeline-unit:1.16' // JUnit testImplementation 'junit:junit:4.13.2' }

**✅ Case 1:** vars/mathStep.groovy **(no Jenkins DSL → no mocks)**

def call(int a, int b) { return a + b }

src/test/groovy/MathStepTest.groovy

import com.lesfurets.jenkins.unit.BasePipelineTest import org.junit.Before import org.junit.Test import static org.junit.Assert.\* class MathStepTest extends BasePipelineTest { @Before void setUp() { super.setUp() } @Test void testAddition() { def script = loadScript("vars/mathStep.groovy") def result = script.call(5, 7) assertEquals(12, result) // ✅ no mocks needed } }

👉 Works like a normal Groovy function test.

**✅ Case 2:** vars/echoStep.groovy **(uses Jenkins DSL → needs mocks)**

def call(String name = "World") { echo "Hello, ${name}!" }

src/test/groovy/EchoStepTest.groovy

import com.lesfurets.jenkins.unit.BasePipelineTest import org.junit.Before import org.junit.Test import static org.junit.Assert.\* class EchoStepTest extends BasePipelineTest { @Before void setUp() { super.setUp() // Mock Jenkins pipeline step "echo" helper.registerAllowedMethod("echo", [String.class], { msg -> println "ECHO: $msg" }) } @Test void testDefaultEcho() { def script = loadScript("vars/echoStep.groovy") script.call() // calls echo inside printCallStack() assertTrue helper.callStack.find { call -> call.methodName == "echo" && callArgsToString(call).contains("Hello, World!") } != null } @Test void testCustomEcho() { def script = loadScript("vars/echoStep.groovy") script.call("Sumanth") assertTrue helper.callStack.find { call -> call.methodName == "echo" && callArgsToString(call).contains("Hello, Sumanth!") } != null } }

👉 Works only if we mock Jenkins DSL steps (echo).

**🔑 Summary**

* Pure Groovy in vars/ (no Jenkins DSL) → ✅ no mocks needed, test like normal Groovy.
* Jenkins DSL in vars/ (echo, sh, git, etc.) → ❌ must mock those steps with JenkinsPipelineUnit.

⚡ Question for you:  
Do you want me to also give an end-to-end Jenkinsfile that uses both mathStep and echoStep from this shared lib, so you can run it inside Jenkins as well?

**You said:**

import com.example.GreetUtils

def call(String name) {

    echo "Preparing to greet ${name}"

    def greeting = GreetUtils.buildGreeting(name)

    echo greeting

    return greeting

}

=======================

import com.lesfurets.jenkins.unit.BasePipelineTest

import org.junit.Before

import org.junit.Test

import static org.junit.Assert.\*

class GreetUserStepTest extends BasePipelineTest {

    @Before

    void setUp() {

        super.setUp()

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

    }

    @Test

    void testGreetUserStep() {

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

        def result = script.call("sumanth")

        printCallStack()

        assertEquals("Hello, Sumanth!", result)

        assertTrue helper.callStack.find { call ->

            call.methodName == "echo" && callArgsToString(call).contains("Hello, Sumanth!")

        } != null

    }

}

=================

reetUserStepTest > testGreetUserStep FAILED

    groovy.lang.MissingMethodException at GreetUserStepTest.groovy:23

**ChatGPT said:**

Ahh 👍 I see what’s happening.  
The error:

groovy.lang.MissingMethodException at GreetUserStepTest.groovy:23

usually means JenkinsPipelineUnit couldn’t find the method/class you’re trying to use in the test.

Let’s break it down:

**Your** vars/greetUser.groovy

import com.example.GreetUtils // 👈 referencing class def call(String name) { echo "Preparing to greet ${name}" def greeting = GreetUtils.buildGreeting(name) echo greeting return greeting }

👉 This expects a class com.example.GreetUtils under src/main/groovy/com/example/GreetUtils.groovy.

**Your Test**

def script = loadScript("vars/greetUser.groovy") def result = script.call("sumanth")

Error at line 23 → that’s the script.call("sumanth") line.  
Which means when running, GreetUtils.buildGreeting() wasn’t found → MissingMethodException.

**✅ Fix**

You need to actually create the utility class. Without it, JenkinsPipelineUnit can’t resolve it.

**📂 `src/main/groovy**

**You said:**

Got this error see for below setup and explain me

GreetUserStepTest > testGreetUserStep FAILED

    groovy.lang.MissingMethodException at GreetUserStepTest.groovy:23

GreetUtilsTest > testGreeting FAILED

    groovy.lang.MissingPropertyException at GreetUtilsTest.groovy:8

GreetUtilsTest > testEmptyNameFails FAILED

    java.lang.Exception at ExpectException.java:30

        Caused by: groovy.lang.MissingPropertyException at GreetUtilsTest.groovy:13

=============================

vars/

import org.example.utils.GreetUtils

def call(String name) {

    echo "Preparing to greet ${name}"

    def greeting = GreetUtils.buildGreeting(name)

    echo greeting

    return greeting

}

================================

/src

package org.example.utils

class GreetUtils {

    static String buildGreeting(String name) {

        if (!name?.trim()) {

            throw new IllegalArgumentException("Name must not be empty")

        }

        return "Hello, ${name.capitalize()}!"

    }

}

=========================

/test both with and without mocking

import com.lesfurets.jenkins.unit.BasePipelineTest

import org.junit.Before

import org.junit.Test

import static org.junit.Assert.\*

class GreetUserStepTest extends BasePipelineTest {

    @Before

    void setUp() {

        super.setUp()

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

    }

    @Test

    void testGreetUserStep() {

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

        def result = script.call("sumanth")

        printCallStack()

        assertEquals("Hello, Sumanth!", result)

        assertTrue helper.callStack.find { call ->

            call.methodName == "echo" && callArgsToString(call).contains("Hello, Sumanth!")

        } != null

    }

}

------------------

import org.junit.Test

import static org.junit.Assert.\*

class GreetUtilsTest {

    @Test

    void testGreeting() {

        assertEquals("Hello, Sumanth!", com.example.GreetUtils.buildGreeting("sumanth"))

    }

    @Test(expected = IllegalArgumentException)

    void testEmptyNameFails() {

        com.example.GreetUtils.buildGreeting("")

    }

}

**ChatGPT said:**

Ahh nice 👍 thanks for pasting the full setup — the errors now make total sense.

Let’s walk through them carefully.

**1.** GreetUserStepTest > testGreetUserStep FAILED MissingMethodException

Your vars/greetUser.groovy:

import org.example.utils.GreetUtils // 👈 package = org.example.utils def call(String name) { echo "Preparing to greet ${name}" def greeting = GreetUtils.buildGreeting(name) echo greeting return greeting }

Your utility class:

package org.example.utils // 👈 matches above import class GreetUtils { static String buildGreeting(String name) { if (!name?.trim()) { throw new IllegalArgumentException("Name must not be empty") } return "Hello, ${name.capitalize()}!" } }

✅ This part is fine.

But in your test GreetUtilsTest, you wrote:

assertEquals("Hello, Sumanth!", com.example.GreetUtils.buildGreeting("sumanth"))

👉 That’s wrong — it’s looking for com.example.GreetUtils, but your class lives in org.example.utils.GreetUtils.

**2.** GreetUtilsTest > MissingPropertyException

That’s because Groovy couldn’t resolve com.example.GreetUtils (wrong package).

**3.** How to Fix

Update your GreetUtilsTest.groovy:

import org.junit.Test import static org.junit.Assert.\* import org.example.utils.GreetUtils // 👈 import the right class class GreetUtilsTest { @Test void testGreeting() { assertEquals("Hello, Sumanth!", GreetUtils.buildGreeting("sumanth")) } @Test(expected = IllegalArgumentException) void testEmptyNameFails() { GreetUtils.buildGreeting("") } }

**4. Why** GreetUserStepTest **failed**

This one blew up at:

def result = script.call("sumanth")

That means when JenkinsPipelineUnit executed greetUser.groovy, it couldn’t resolve GreetUtils → because of the same package mismatch problem.

Once you fix the package usage (make tests + imports consistently use org.example.utils.GreetUtils), this will work.

**✅ Final Checklist**

* vars/greetUser.groovy
* import org.example.utils.GreetUtils
* src/main/groovy/org/example/utils/GreetUtils.groovy
* package org.example.utils
* tests should either import org.example.utils.GreetUtils or call it fully qualified.

⚡ In short:  
Your errors come from package mismatch (com.example vs org.example.utils).  
Fix all references to org.example.utils.GreetUtils and your tests will pass.

👉 Do you want me to rewrite the whole vars/, src/, and both test files with the fixed package structure so you can copy-paste and run cleanly?

**You said:**

GreetUserStepTest > testGreetUserStep FAILED

    groovy.lang.MissingMethodException at GreetUserStepTest.groovy:23

====

groovy.lang.MissingMethodException: No signature of method: GreetUserStepTest.callArgsToString() is applicable for argument types: (com.lesfurets.jenkins.unit.MethodCall) values: [      greetUser.echo(Preparing to greet sumanth)]

at app//org.codehaus.groovy.runtime.ScriptBytecodeAdapter.unwrap(ScriptBytecodeAdapter.java:70)

at app//org.codehaus.groovy.runtime.callsite.PogoMetaClassSite.callCurrent(PogoMetaClassSite.java:80)

at app//org.codehaus.groovy.runtime.callsite.CallSiteArray.defaultCallCurrent(CallSiteArray.java:51)

at app//org.codehaus.groovy.runtime.callsite.AbstractCallSite.callCurrent(AbstractCallSite.java:171)

at app//org.codehaus.groovy.runtime.callsite.AbstractCallSite.callCurrent(AbstractCallSite.java:185)

at app//GreetUserStepTest$\_testGreetUserStep\_closure2.doCall(GreetUserStepTest.groovy:24)

at java.base@17.0.16/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at java.base@17.0.16/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:77)

at java.base@17.0.16/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at java.base@17.0.16/java.lang.reflect.Method.invoke(Method.java:569)

at app//org.codehaus.groovy.reflection.CachedMethod.invoke(CachedMethod.java:107)

at app//groovy.lang.MetaMethod.doMethodInvoke(MetaMethod.java:323)

at app//org.codehaus.groovy.runtime.metaclass.ClosureMetaClass.invokeMethod(ClosureMetaClass.java:274)

at app//groovy.lang.MetaClassImpl.invokeMethod(MetaClassImpl.java:1030)

at app//org.codehaus.groovy.runtime.callsite.PogoMetaClassSite.call(PogoMetaClassSite.java:38)

at app//org.codehaus.groovy.runtime.callsite.BooleanReturningMethodInvoker.invoke(BooleanReturningMethodInvoker.java:49)

at app//org.codehaus.groovy.runtime.callsite.BooleanClosureWrapper.call(BooleanClosureWrapper.java:52)

at app//org.codehaus.groovy.runtime.DefaultGroovyMethods.find(DefaultGroovyMethods.java:4453)

at app//org.codehaus.groovy.runtime.dgm$237.invoke(Unknown Source)

at app//org.codehaus.groovy.runtime.callsite.PojoMetaMethodSite$PojoMetaMethodSiteNoUnwrapNoCoerce.invoke(PojoMetaMethodSite.java:242)

at app//org.codehaus.groovy.runtime.callsite.PojoMetaMethodSite.call(PojoMetaMethodSite.java:51)

at app//org.codehaus.groovy.runtime.callsite.CallSiteArray.defaultCall(CallSiteArray.java:47)

at app//org.codehaus.groovy.runtime.callsite.AbstractCallSite.call(AbstractCallSite.java:125)

at app//org.codehaus.groovy.runtime.callsite.AbstractCallSite.call(AbstractCallSite.java:139)

at app//GreetUserStepTest.testGreetUserStep(GreetUserStepTest.groovy:23)

at java.base@17.0.16/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at java.base@17.0.16/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:77)

at java.base@17.0.16/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at java.base@17.0.16/java.lang.reflect.Method.invoke(Method.java:569)

at app//org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:59)

at app//org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)

at app//org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:56)

at app//org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)

at app//org.junit.internal.runners.statements.RunBefores.evaluate(RunBefores.java:26)

at app//org.junit.runners.ParentRunner$3.evaluate(ParentRunner.java:306)

at app//org.junit.runners.BlockJUnit4ClassRunner$1.evaluate(BlockJUnit4ClassRunner.java:100)

at app//org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:366)

at app//org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:103)

at app//org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:63)

at app//org.junit.runners.ParentRunner$4.run(ParentRunner.java:331)

at app//org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:79)

at app//org.junit.runners.ParentRunner.runChildren(ParentRunner.java:329)

at app//org.junit.runners.ParentRunner.access$100(ParentRunner.java:66)

at app//org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:293)

at app//org.junit.runners.ParentRunner$3.evaluate(ParentRunner.java:306)

at app//org.junit.runners.ParentRunner.run(ParentRunner.java:413)

at org.gradle.api.internal.tasks.testing.junit.JUnitTestClassExecutor.runTestClass(JUnitTestClassExecutor.java:112)

at org.gradle.api.internal.tasks.testing.junit.JUnitTestClassExecutor.execute(JUnitTestClassExecutor.java:58)

at org.gradle.api.internal.tasks.testing.junit.JUnitTestClassExecutor.execute(JUnitTestClassExecutor.java:40)

at org.gradle.api.internal.tasks.testing.junit.AbstractJUnitTestClassProcessor.processTestClass(AbstractJUnitTestClassProcessor.java:54)

at org.gradle.api.internal.tasks.testing.SuiteTestClassProcessor.processTestClass(SuiteTestClassProcessor.java:53)

at java.base@17.0.16/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at java.base@17.0.16/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:77)

at java.base@17.0.16/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at java.base@17.0.16/java.lang.reflect.Method.invoke(Method.java:569)

at org.gradle.internal.dispatch.ReflectionDispatch.dispatch(ReflectionDispatch.java:36)

at org.gradle.internal.dispatch.ReflectionDispatch.dispatch(ReflectionDispatch.java:24)

at org.gradle.internal.dispatch.ContextClassLoaderDispatch.dispatch(ContextClassLoaderDispatch.java:33)

at org.gradle.internal.dispatch.ProxyDispatchAdapter$DispatchingInvocationHandler.invoke(ProxyDispatchAdapter.java:92)

at jdk.proxy2/jdk.proxy2.$Proxy6.processTestClass(Unknown Source)

at org.gradle.api.internal.tasks.testing.worker.TestWorker$2.run(TestWorker.java:181)

at org.gradle.api.internal.tasks.testing.worker.TestWorker.executeAndMaintainThreadName(TestWorker.java:130)

at org.gradle.api.internal.tasks.testing.worker.TestWorker.execute(TestWorker.java:101)

at org.gradle.api.internal.tasks.testing.worker.TestWorker.execute(TestWorker.java:61)

at org.gradle.process.internal.worker.child.ActionExecutionWorker.execute(ActionExecutionWorker.java:56)

at org.gradle.process.internal.worker.child.SystemApplicationClassLoaderWorker.call(SystemApplicationClassLoaderWorker.java:122)

at org.gradle.process.internal.worker.child.SystemApplicationClassLoaderWorker.call(SystemApplicationClassLoaderWorker.java:69)

at app//worker.org.gradle.process.internal.worker.GradleWorkerMain.run(GradleWorkerMain.java:69)

at app//worker.org.gradle.process.internal.worker.GradleWorkerMain.main(GradleWorkerMain.java:74)

again for this one?

**ChatGPT said:**

Perfect, thanks for sharing the full error 🙏  
This one is different from the earlier package mismatch.

**🔎 Root Cause**

Your stacktrace shows:

groovy.lang.MissingMethodException: No signature of method: GreetUserStepTest.callArgsToString()

That means:  
👉 In your test, you called callArgsToString(call) but callArgsToString is not a method in your test class.

**🛠 Why?**

callArgsToString is a helper method from JenkinsPipelineUnit examples, but it’s not automatically available in your BasePipelineTest class unless you implement it yourself.

That’s why Groovy thinks you’re trying to call a method callArgsToString() on GreetUserStepTest, which doesn’t exist → MissingMethodException.

**✅ Two Ways to Fix**

**Option 1: Don’t use** callArgsToString

You can directly access the call arguments from the call stack.  
Replace:

assertTrue helper.callStack.find { call -> call.methodName == "echo" && callArgsToString(call).contains("Hello, Sumanth!") } != null

with:

assertTrue helper.callStack.find { call -> call.methodName == "echo" && call.args[0].contains("Hello, Sumanth!") } != null

**Option 2: Implement your own** callArgsToString

If you like the shorthand, define this method in your test:

String callArgsToString(def call) { return call.args.collect { it.toString() }.join(", ") }

Then your original code with callArgsToString(call) will work.

**🧪 Final Working Test**

import com.lesfurets.jenkins.unit.BasePipelineTest import org.junit.Before import org.junit.Test import static org.junit.Assert.\* class GreetUserStepTest extends BasePipelineTest { @Before void setUp() { super.setUp() helper.registerAllowedMethod("echo", [String.class], { s -> println "ECHO: $s" }) } @Test void testGreetUserStep() { def script = loadScript("vars/greetUser.groovy") def result = script.call("sumanth") printCallStack() assertEquals("Hello, Sumanth!", result) // ✅ fixed assertTrue helper.callStack.find { call -> call.methodName == "echo" && call.args[0].contains("Hello, Sumanth!") } != null } }

⚡ In short:

* The error wasn’t about Jenkins or Groovy classpath.
* It’s just that callArgsToString() doesn’t exist in your test.
* Use call.args[0] directly, or implement that helper.

👉 Do you want me to also show you a minimal no-mocking test for the same vars/greetUser.groovy (just unit test the Groovy logic directly, without JenkinsPipelineUnit)?

**You said:**

@Library('jenkins-shared-library') \_

pipeline {

    agent any   // Runs on EC2 host

    options {

        timestamps()

    }

    stages {

        stage('Unit Tests') {

            steps {

                // Run Gradle wrapper tests

                sh """

                chmod +x gradlew

                ./gradlew clean test

                """

            }

            post {

                always {

                    junit 'build/test-results/test/\*.xml'

                }

            }

        }

        stage('Code Coverage') {

            steps {

                sh './gradlew jacocoTestReport'

            }

            post {

                always {

                    publishHTML(target: [

                        reportDir: 'build/reports/jacoco/test/html',

                        reportFiles: 'index.html',

                        reportName: 'JaCoCo Coverage'

                    ])

                }

            }

        }

        stage('SonarQube Analysis') {

            steps {

                withSonarQubeEnv('SonarQubeServer') {

                    sh './gradlew sonarqube'

                }

            }

        }

    }

}

anything is problem with this what and i need to do to get everything working in jenkins

**ChatGPT said:**

Good one 👍 you’re now combining Gradle + Jacoco + SonarQube inside a Jenkins pipeline.  
Let’s review it carefully and see what could go wrong and what you need to set up ✅

**🔎 Things to check**

**1. Shared Library Import**

@Library('jenkins-shared-library') \_

* This assumes you have a global pipeline library configured in Jenkins (Manage Jenkins → Configure System → Global Pipeline Libraries).
* If you’re not using any library functions inside this pipeline, you can remove this line safely.
* If you *are* using it, make sure:
  + The Git repo is configured.
  + Branch (usually master or main) is correct.
  + It’s under Global Untrusted Pipeline Libraries unless you’re an admin and want to allow trusted execution.

**2. Gradle Wrapper (**gradlew**)**

sh """ chmod +x gradlew ./gradlew clean test """

* ✅ Correct usage.
* Make sure gradlew exists in your repo.
* If your EC2 has Java 21 installed, Gradle wrapper must be a version compatible with Java 21 (Gradle 8.3+ required). You said you’re on Gradle 8.13 → that’s fine.

**3. JUnit Publishing**

junit 'build/test-results/test/\*.xml'

* ✅ Works if your Gradle is using testImplementation and default junit task.
* Ensure your build.gradle has the JUnit platform (e.g., JUnit4/5).
* Jenkins must have the JUnit plugin installed.

**4. Code Coverage (JaCoCo)**

sh './gradlew jacocoTestReport'

* ✅ Correct if you have this in build.gradle:
* plugins { id 'jacoco' } jacocoTestReport { reports { xml.required = true html.required = true } }
* Also install the HTML Publisher Plugin in Jenkins.
* Otherwise, the publishHTML step will fail.

**5. SonarQube Analysis**

withSonarQubeEnv('SonarQubeServer') { sh './gradlew sonarqube' }

* ✅ Correct, but requires:
  + SonarQube Scanner for Jenkins plugin installed.
  + SonarQube server configured in Jenkins (Manage Jenkins → Configure System → SonarQube Servers).
  + Your build.gradle must include:
  + plugins { id "org.sonarqube" version "5.0.0.4638" // or latest }
  + And properties (sonar-project.properties file or build.gradle config):
  + sonarqube { properties { property "sonar.projectKey", "your-project-key" property "sonar.host.url", "http://<sonarqube-server>:9000" property "sonar.login", "token" // optional if using Jenkins token } }

**6. EC2 Agent**

* Your Jenkins EC2 node must have:
  + docker (if you’re using docker-based builds).
  + java --version = 21 (or what your Gradle project requires).
  + Internet access (for Gradle dependencies and SonarQube uploads).
  + Gradle wrapper (gradlew) inside your repo. Don’t rely on system Gradle.

**🚀 Full Pipeline Recap**

Your pipeline is fine structurally. To make it work end-to-end you need:

✅ Plugins installed in Jenkins:

* JUnit
* HTML Publisher
* SonarQube Scanner for Jenkins

✅ Gradle setup in repo:

* gradlew script checked in
* jacoco plugin applied
* sonarqube plugin applied

✅ Jenkins setup:

* SonarQube server defined in Manage Jenkins → Configure System → SonarQube servers
* (Optional) Jenkins Global Shared Library if you keep @Library

👉 Question for you:  
Do you want me to give you a working build.gradle template (Java 21, Gradle 8.13, JUnit5, JaCoCo, SonarQube) so that this pipeline runs out-of-the-box?

**You said:**

Give me full documentation type to explain why we are doing what we are doing and it should not be like college presentation and it should be POC submission at work where we will not miss important things at the same time we will not go very boring as well

**ChatGPT said:**

Got it ✅ — you want something like a POC documentation for your Jenkins pipeline setup that explains the *why* behind each step (not just the *what*), with a balance between professional + practical. Below is a draft you could use as a POC submission at work.

**POC: Jenkins CI Pipeline with Unit Tests, Code Coverage, and SonarQube Analysis**

**Objective**

The purpose of this Proof of Concept (POC) is to validate a Jenkins CI pipeline that ensures code quality by automating:

1. Unit Testing – Validate correctness of business logic.
2. Code Coverage Reporting (JaCoCo) – Measure test coverage for better visibility.
3. Static Code Analysis (SonarQube) – Detect code smells, security issues, and maintainability risks.

This pipeline is implemented as a declarative Jenkinsfile, running in an EC2-hosted Jenkins agent with Gradle as the build tool.

**Pipeline Design**

**Stage 1: Unit Tests**

stage('Unit Tests') { steps { sh """ chmod +x gradlew ./gradlew clean test """ } post { always { junit 'build/test-results/test/\*.xml' } } }

* Why?
  + Ensures all unit tests run on every commit/pipeline run.
  + gradlew clean test guarantees a fresh build (no stale results).
  + Publishing JUnit reports integrates test results into Jenkins UI.
* What this gives us in Jenkins:  
  ✅ Red/green pipeline status based on tests  
  ✅ Visible pass/fail counts  
  ✅ Historical trends for test stability

**Stage 2: Code Coverage**

stage('Code Coverage') { steps { sh './gradlew jacocoTestReport' } post { always { publishHTML(target: [ reportDir: 'build/reports/jacoco/test/html', reportFiles: 'index.html', reportName: 'JaCoCo Coverage' ]) } } }

* Why?
  + Knowing what percentage of code is tested prevents shipping untested features.
  + JaCoCo integrates directly with Gradle, producing both XML (machine-readable) and HTML (human-readable) reports.
  + Reports are published in Jenkins for easy team access.
* What this gives us in Jenkins:  
  ✅ HTML coverage reports embedded in Jenkins UI  
  ✅ Team visibility on coverage trends  
  ✅ Opportunity to set coverage thresholds in future (e.g., fail pipeline if < 70%)

**Stage 3: SonarQube Analysis**

stage('SonarQube Analysis') { steps { withSonarQubeEnv('SonarQubeServer') { sh './gradlew sonarqube' } } }

* Why?
  + SonarQube provides deep static code analysis:
    - Code smells
    - Security vulnerabilities
    - Maintainability issues
    - Duplications
  + Running this as part of CI enforces quality gates early.
* Key Jenkins config required:
  + A SonarQube server (SonarQubeServer) must be configured in Manage Jenkins → System Config.
  + sonar-project.properties or Gradle SonarQube plugin must be set in project.
* What this gives us in Jenkins:  
  ✅ Analysis results pushed to SonarQube dashboard  
  ✅ Pipeline can fail on quality gate failures (optional)  
  ✅ Trends of code health across commits

**Prerequisites**

1. Infrastructure
   * Jenkins deployed on EC2 (Docker or native install).
   * Java & Gradle available (or wrapper checked into repo).
2. Jenkins Setup
   * Plugins: JUnit, HTML Publisher, SonarQube Scanner, Pipeline.
   * Global Tool Config: JDK, Git, Gradle (optional if wrapper used).
   * SonarQube server connection configured.
3. Project Setup
   * gradlew wrapper script included and executable.
   * build.gradle includes:
     + jacoco plugin
     + sonarqube plugin
     + testImplementation dependencies

**Benefits of This Approach**

* Shift Left Testing → Bugs caught during development, not in production.
* Visibility & Accountability → Test results, coverage, and analysis are shared with the team.
* Standardization → Common pipeline across services ensures consistent quality checks.
* Future Extensions Possible →
  + Coverage thresholds (fail if < X%)
  + SonarQube quality gates as pipeline blockers
  + Artifact upload (JFrog, Nexus)
  + Deployment stages

**Sample Output in Jenkins**

* Unit Test Stage → ECHO: Preparing to greet sumanth + JUnit report.
* Coverage Stage → HTML report with % coverage breakdown.
* SonarQube Stage → "Quality Gate Passed/Failed" status in Jenkins.

**Conclusion**

This POC demonstrates a CI-first mindset: before thinking about deployments, we ensure our code is tested, measured, and analyzed. By enforcing this pipeline across projects, we establish baseline code quality practices and reduce risks in later environments (TEST/PROD).

👉 Would you like me to also add example build.gradle configuration for JaCoCo + SonarQube (so the POC is fully runnable end-to-end)?

**You said:**

still feels like some step by step explaination can we like give a overvieww of why we are going in this aproach and what are all the required setup to do a jenkinspipelineunit testing for shared libraries. at the same time what is very critical resources(setup), procedure if we have to test /vars , /src/ with mocking without mocking all these things.....both give it to test in local and jenkins testing also try to add if I am missing anything

Don't need a exact set up like i need the document like explaination

**ChatGPT said:**

Thought for 21s

**Jenkins Shared Library — Practical Testing Guide (POC-style)**

This is the “why + how” guide we’d hand to teammates when we want everyone to build and test Jenkins Shared Libraries the same way—without over-explaining and without missing what matters.

**1) Why this approach?**

* Separation of concerns
  + vars/ = thin Jenkins-facing wrappers (DSL: echo, sh, checkout, etc.).
  + src/main/groovy = pure, reusable logic (no Jenkins DSL).
* Testability
  + Pure logic is testable with plain JUnit (no Jenkins dependency, no mocks).
  + DSL wrappers are testable with JenkinsPipelineUnit (mock only what you use).
* Speed & Safety
  + Fast local feedback (Gradle + JUnit).
  + Deterministic tests (no real shells, no real Git, no network calls).
* Portability
  + Works locally and in Jenkins agents the same way (via the Gradle Wrapper).

**2) What are we testing (exactly)?**

* Unit tests for src/ (no Jenkins)
  + Business logic, validation, formatting, small utilities.
  + Example: map environment → URL, string formatting, JSON parsing, etc.
* Unit tests for vars/ (Jenkins DSL)
  + Did we call the right Jenkins steps with the right arguments?
  + Use JenkinsPipelineUnit’s helper.callStack to assert behavior.
* What we are *not* testing
  + Real sh execution, real Git checkouts, real credentials—those are integration concerns.

**3) What do we need (critical setup)**

**Local workstation**

* Java: 21 (to match the project)
* Gradle Wrapper: 8.1.3+ (you said 8.13; good)
* Dependencies (in Gradle):
  + groovy (implementation localGroovy()),
  + com.lesfurets:jenkins-pipeline-unit for DSL tests,
  + junit (4 or 5),
  + optional: assertj/hamcrest for nicer assertions.
* Directory layout (conventional):

jenkins-shared-lib/

├─ vars/              *# Jenkins-facing “steps”: def call(...) { ... }*

├─ src/main/groovy/   *# Pure Groovy classes, packages*

└─ src/test/groovy/   *# Unit tests for both*

**Jenkins controller/agent**

* Agent runtime: Java 21, Git, internet egress for Gradle deps.
* Plugins: Pipeline, JUnit, HTML Publisher (for coverage report), SonarQube Scanner (if used).
* Global Pipeline Library (if you want to import with @Library('name')):
  + Configure in Manage Jenkins → Configure System → Global Pipeline Libraries.
  + Point to your Git repo + default branch.
* Permissions: If you use Trusted vs Untrusted libraries, decide per security posture.

**4) How to decide: mock or not?**

Use this decision tree:

* Does the code call Jenkins DSL (echo, sh, git, checkout, withCredentials, etc.)?
  + Yes → test with JenkinsPipelineUnit and mock those steps.
  + No → test with plain JUnit, no mocks.
* Is the code in vars/ but pure Groovy (no DSL)?
  + You can still load it via JenkinsPipelineUnit and run without mocks.
  + Or move the logic into src/ and unit test with plain JUnit (preferred).

Recommended pattern (production)

* Keep vars/ tiny. Push logic into src/.
* Then you only mock thin wrappers, and you test the real logic without mocks.

**5) Procedures (concise, copy-paste checklists)**

**A) Testing** src/ **(pure Groovy) — without mocks**

Goal: Fast, deterministic tests for business logic.

* Put classes under src/main/groovy/{your package}/…
* Write JUnit tests under src/test/groovy/…
* Run locally: ./gradlew clean test
* In Jenkins: run the same command and publish build/test-results/test/\*.xml.
* If using coverage: add jacoco plugin and run ./gradlew jacocoTestReport (publish HTML).

When this fails, it’s usually:

* Package mismatch (package declaration vs folder path).
* Missing dependency for the logic (e.g., JSON lib).
* Using Jenkins DSL by accident in src/ (don’t).

**B) Testing** vars/ **(uses Jenkins DSL) — with mocks**

Goal: Verify the wrapper calls the right Jenkins steps.

* Extend BasePipelineTest in tests.
* In @Before setUp():
  + super.setUp()
  + helper.registerAllowedMethod("echo", [String]) { ... }
  + Register *every* DSL you call (sh, echo, withCredentials, etc.).
* Load script with loadScript("vars/yourStep.groovy") and call it.
* Assert using helper.callStack (e.g., call.methodName == "sh" and call.args[0] contains "...").
* Run locally + Jenkins the same way (./gradlew test).

Common gotchas:

* Forgetting to register a DSL method → MissingMethodException.
* Using helper utilities that aren’t provided (e.g., callArgsToString) → access call.args[n] directly.
* vars/ importing a class whose package doesn’t match the folder path → MissingProperty/MethodException.

**C) Testing** vars/ **(pure Groovy) — without mocks**

Goal: You *can* do this if vars/ uses no DSL. It’s the exception, not the rule.

* Still load via loadScript("vars/foo.groovy").
* Call and assert the return value / side effects.
* If you need DSL tomorrow, this test will start failing → better to move logic to src/ early.

**6) Running locally vs in Jenkins**

**Local**

* ./gradlew clean test
* Optional: ./gradlew jacocoTestReport
* Open coverage locally: build/reports/jacoco/test/html/index.html

**Jenkins (declarative pipeline)**

* Unit Tests: run wrapper: ./gradlew clean test, then junit 'build/test-results/test/\*.xml'
* Coverage: ./gradlew jacocoTestReport, then publish HTML report.
* Optional SonarQube: with credentials configured, run ./gradlew sonarqube inside withSonarQubeEnv(...).

Heads-up:

* Keep the Gradle Wrapper in the repo. Don’t rely on global Gradle.
* Ensure agent Java matches your project (Java 21).
* If builds are slow due to cold caches, consider Gradle build cache or a persistent worker.

**7) Troubleshooting “classics” (fast answers)**

* MissingMethodException in a vars/ test
  + You called a Jenkins DSL step you didn’t register (e.g., forgot to register sh).
  + Or you referenced a helper like callArgsToString that you never implemented.
* MissingPropertyException or wrong class in tests
  + Package mismatch. The package in the source file must match the folder path.
  + In tests, import the right package (e.g., org.example.utils.GreetUtils).
* groovy.lang.MissingMethodException: No signature of method: echo()
  + You’re running a test without mocking DSL. Register echo in the test setUp().
* Tests pass locally, fail in Jenkins
  + Jenkins agent uses a different Java version.
  + Missing plugins (JUnit/HTML Publisher).
  + Jenkins workspace path issues (file path assumptions in tests).
* JaCoCo report not visible
  + HTML Publisher plugin not installed.
  + Wrong reportDir path.
* SonarQube step fails
  + Server not configured in Jenkins, or token missing.
  + Sonar plugin not applied in Gradle.

**8) Quality & guardrails we can add later**

* Coverage thresholds
  + Fail the build if coverage < X% (JaCoCo + Gradle check task).
* Sonar Quality Gate
  + Wait for Sonar’s quality gate and fail pipeline if it fails.
* Pre-commit hooks
  + Run ./gradlew test locally before commit.
* Library versioning
  + Tag releases of the shared lib; pipelines consume a specific tag/branch.
* API discipline for vars/
  + Keep signatures stable; document parameters and expected behavior.

**9) Quick reference (pin-on-the-wall)**

* Put logic in src/ → JUnit tests, zero mocks.
* Keep vars/ thin → if DSL is used, mock in tests; assert via helper.callStack.
* Local runs and Jenkins runs should both use ./gradlew.
* Register every DSL you call in vars/ tests.
* Don’t fight packages → folder path ↔ package declaration must align.
* When in doubt: if it touches Jenkins, mock it. If it doesn’t, plain JUnit.

If you want, I can convert this into a short team-facing Confluence page with a one-page checklist plus a “Known issues & fixes” section.

**You said:**

See I am not against mocking the classes in jenkins

I do have to present this as well but how is like how you said yesterday -->

The Correct Approach for JPU

JenkinsPipelineUnit cannot reliably mock constructors. The tested class must accept dependency injection. That is, your deployApp.groovy must be slightly refactored:

or is there any way different for for jenkinspipelineunit for jenkins shared library includeing /vars, /src or if required /resources

**ChatGPT said:**

Got it 👍 You’re asking an architectural-level clarification:  
*When testing Jenkins Shared Libraries with JenkinsPipelineUnit (JPU), is the “dependency injection refactor” the only correct approach? Or are there alternative strategies, especially when we’re dealing with /vars, /src, /resources?*

Let me break it down cleanly and realistically (this is how teams typically document it when setting up POC/standards for Jenkins Shared Library testing).

**🌍 Big Picture**

Jenkins Shared Libraries are not normal Groovy apps – they’re DSL wrappers that get executed inside Jenkins.

* vars/ → pipeline steps (e.g. deployApp.groovy)
* src/ → utility classes (reusable pure Groovy/Java code)
* resources/ → static templates, configs, etc.

When testing with JenkinsPipelineUnit (JPU):

* The DSL (pipeline {}, echo, sh, etc.) must be mocked because you can’t really call Jenkins runtime in unit tests.
* Pure utilities in src/ should be tested like normal Groovy/Java with JUnit (no mocking needed unless you call Jenkins DSL inside).

**⚡ The “Problem” with Constructors and Jenkins DSL**

JPU does not support constructor-level mocking well. Example:

class Deployer { JenkinsHelper helper = new JenkinsHelper() // <- tightly coupled void deploy() { helper.sh("deploy script") // real DSL call } }

If you test this:

* JPU can’t intercept the new JenkinsHelper() inside the class → your test either fails or tries to call a real DSL.

**✅ Correct Fix = Dependency Injection**

Refactor to accept dependencies:

class Deployer { JenkinsHelper helper Deployer(JenkinsHelper helper) { this.helper = helper } void deploy() { helper.sh("deploy script") } }

Now in the test:

def mockHelper = Mock(JenkinsHelper) def deployer = new Deployer(mockHelper) deployer.deploy()

This works because you control the collaborator.

**🎯 Alternative Approaches (besides DI)**

There *are* alternatives, but each comes with trade-offs:

**1. Direct Mock Registration (works best for** vars/**)**

helper.registerAllowedMethod("sh", [String.class], { s -> println "MOCK: $s" })

* Works well for vars/steps.groovy.
* But if your /src class creates objects internally → mocking fails, hence DI is still cleaner.

**2. Partial Mocking with Groovy Meta-programming**

Groovy allows you to override methods at runtime:

JenkinsHelper.metaClass.sh = { String cmd -> println "Intercepted $cmd" }

* Can sometimes “save” legacy code without DI.
* But brittle: metaClass can leak into other tests, leading to flaky builds.
* Teams usually avoid this in production-grade testing frameworks.

**3. Use** resources/ **for Static DSL Injection**

Sometimes you offload DSL-heavy logic to a resource (like a shell script or JSON template). Example:

def script = libraryResource('deploy.sh') sh(script)

* Test bypasses sh call entirely → you only test resource resolution.
* Cleaner separation of concerns, but doesn’t solve constructor mocking.

**4. End-to-End (Smoke) Testing in Jenkins Itself**

You can also skip mocking altogether for critical steps:

* Run a real Jenkins pipeline job pointing at your shared library.
* Validate the vars/ step works in real environment (but slower and not unit-test level).

This complements JPU tests, not replaces them.

**🧩 Where Each Strategy Fits**

|  |  |  |  |
| --- | --- | --- | --- |
| **Location** | **Testing Strategy** | **Mock Needed?** | **Why** |
| /vars (pipeline steps) | JPU + helper.registerAllowedMethod | ✅ | DSL-heavy, so mocking required |
| /src (utilities) | Pure JUnit (no JPU) | ❌ (unless DSL inside) | Should be clean business logic |
| /src with DSL | Refactor for DI (preferred) OR metaClass hack | ✅ | Keeps tests stable |
| /resources | Load + assert content | ❌ | Just test resolution, no DSL |
| Full pipeline | Jenkins job with shared lib loaded | ❌ | Integration/smoke test |

**🚦 Critical Resources You Must Set Up**

1. Local Dev
   * Gradle/Maven build with JUnit + JPU
   * Test runner (./gradlew test)
   * Coverage (jacoco)
   * Mock registration helpers
2. CI/Jenkins
   * Pipeline that runs:
     + ./gradlew clean test (unit tests)
     + jacocoTestReport (coverage)
     + SonarQube (quality gates)
   * Optionally: pipeline job that runs a real smoke test of the shared lib

**🏆 The Real Answer (if you present this at work)**

* For /vars: mock DSL methods with JPU.
* For /src: write pure JUnit tests.
* For /src that depends on Jenkins DSL: refactor to use DI (because JPU cannot reliably mock constructors).
* For /resources: test static resolution.
* For system confidence: run integration job in Jenkins itself.

So yes → the DI approach is the correct & sustainable solution in production,  
but mock registration and metaClass hacks are acceptable stop-gaps for POC/legacy code.

👉 Would you like me to put this into a POC-style doc format (like a lightweight standard your team could adopt), or keep it as an explainer?

**You said:**

jpu means

**ChatGPT said:**

JPU just means JenkinsPipelineUnit ✅

It’s the testing framework (library) that we use to unit test Jenkins Shared Libraries locally without running a real Jenkins instance.

Basically:

* What it is → A Groovy/JUnit based testing framework created by LesFurets.
* Why it exists → Because running pipeline { ... } scripts or shared libraries directly in Jenkins is slow and tied to Jenkins runtime. JPU lets you simulate Jenkins steps (echo, sh, checkout, etc.) and verify the behavior offline.
* Where it fits → It’s the foundation for writing unit tests for your vars/, src/, and even resources/ logic before deploying them to Jenkins.
* How it works → By providing a mocked Jenkins environment (BasePipelineTest) where you can load scripts, register allowed methods, and inspect the call stack.

So whenever we’re talking about “JPU approach” or “JPU tests”, we mean using JenkinsPipelineUnit to test Jenkins Shared Libraries.

👉 If you want, I can break down the right way to handle /vars, /src, and /resources testing with JPU (mocking vs no mocking, dependency injection vs direct call). Would you like me to draft that structure?

**You said:**

Would you like me to put this into a POC-style doc format (like a lightweight standard your team could adopt),

yes give me

**ChatGPT said:**

Got it ✅  
Here’s a POC-style document you could use with your team. It’s not a “college-style” presentation, but a practical explainer that justifies *why* and *how* to do Jenkins Shared Library unit testing with JenkinsPipelineUnit (JPU).

**📝 POC Document: Unit Testing Jenkins Shared Libraries with JenkinsPipelineUnit (JPU)**

**1. Why This Approach?**

Shared Libraries are the backbone of Jenkins pipelines across projects. But:

* Writing logic directly in Jenkins (pipeline {}) is slow, brittle, and hard to debug.
* Deploying broken shared libraries can block multiple teams.
* Jenkins itself is not a great test environment — it’s designed to run pipelines, not debug them.

👉 Solution: Use JenkinsPipelineUnit (JPU) to test shared libraries *locally and in Jenkins CI* before deploying them.  
This gives:

* Fast feedback → no need to spin up Jenkins for every change.
* Confidence → detect syntax errors, wrong steps, and missing arguments early.
* Scalability → same testing approach works for /vars, /src, and /resources.

**2. Critical Setup Required**

**Tools & Dependencies**

* Gradle 8.x (or Maven if preferred)
* Java 21 (LTS, ensures latest features)
* Groovy 3.x+ (same as Jenkins uses internally)
* JUnit 4/5 (test runner)
* JenkinsPipelineUnit (com.lesfurets:jenkins-pipeline-unit)

**Repo Structure (Shared Library)**

jenkins-shared-library/

 ├── vars/               *# Global pipeline steps (Groovy scripts)*

 │    └── deployApp.groovy

 ├── src/                *# Pure Groovy classes (logic, utils)*

 │    └── org/example/Helper.groovy

 ├── resources/          *# Templates, JSON, YAML configs*

 │    └── email-template.txt

 ├── test/               *# JPU-based unit tests*

 │    └── DeployAppTest.groovy

 ├── build.gradle

 └── settings.gradle

**Jenkins Setup**

* Configure library under Manage Jenkins → Global Pipeline Libraries.
* Ensure your Jenkins jobs can run ./gradlew test.
* (Optional) Add a pipeline stage for unit testing shared library before using it.

**3. Testing** /vars**,** /src**,** /resources

**3.1** /vars/ **Files**

* These are entry points (e.g., deployApp.groovy) callable directly in pipelines.
* JPU lets you load and call them like regular Groovy scripts.

With Mocking

class DeployAppTest extends BasePipelineTest { @Before void setUp() { super.setUp() } @Test void testDeployApp() { def script = loadScript("vars/deployApp.groovy") script.call(env: "dev") // Verify Jenkins steps were called assertJobStatusSuccess() assertStepCalls("sh", 1) // ensures sh was invoked once } }

Without Mocking (Pure Logic)  
👉 Only works if your vars/ file pushes logic into src/.

def helper = new org.example.Helper() assert helper.sayHello("Dev") == "Hello Dev"

🔑 Best Practice: Keep /vars thin (only orchestrates Jenkins steps) and push business logic into /src for easier testing.

**3.2** /src/ **Classes**

* Pure Groovy → can be tested without mocking.
* JPU not strictly required, but you may combine both for consistency.

Example:

class HelperTest { @Test void testSayHello() { def helper = new org.example.Helper() assert helper.sayHello("Dev") == "Hello Dev" } }

**3.3** /resources/ **Files**

* Templates, configs, or JSON data.
* Load them in tests to ensure correctness.

@Test void testTemplateLoading() { def template = new File("resources/email-template.txt").text assert template.contains("Subject:") }

**4. Mocking vs No-Mocking**

|  |  |  |
| --- | --- | --- |
| **Case** | **With Mocking (JPU)** | **Without Mocking (Plain Groovy)** |
| /vars/ pipeline steps | ✅ Needed (e.g., sh, git) | ❌ Not possible (relies on Jenkins runtime) |
| /src/ classes | Optional | ✅ Best (pure Groovy logic) |
| /resources/ templates | Not needed | ✅ Direct file test |

👉 Rule of Thumb:

* Pipeline orchestration (vars/) → test with JPU (mock Jenkins steps).
* Business logic (src/) → test without mocking.
* Config/templates (resources/) → test with plain file assertions.

**5. Local vs Jenkins Testing**

**Local (Developer Machine)**

* Run ./gradlew test
* Fast feedback
* JPU simulates Jenkins environment

**Jenkins CI**

* Add Unit Test stage in the shared library Jenkinsfile
* Run the same ./gradlew test
* Publish test reports (junit + jacoco)

**6. Critical Takeaways**

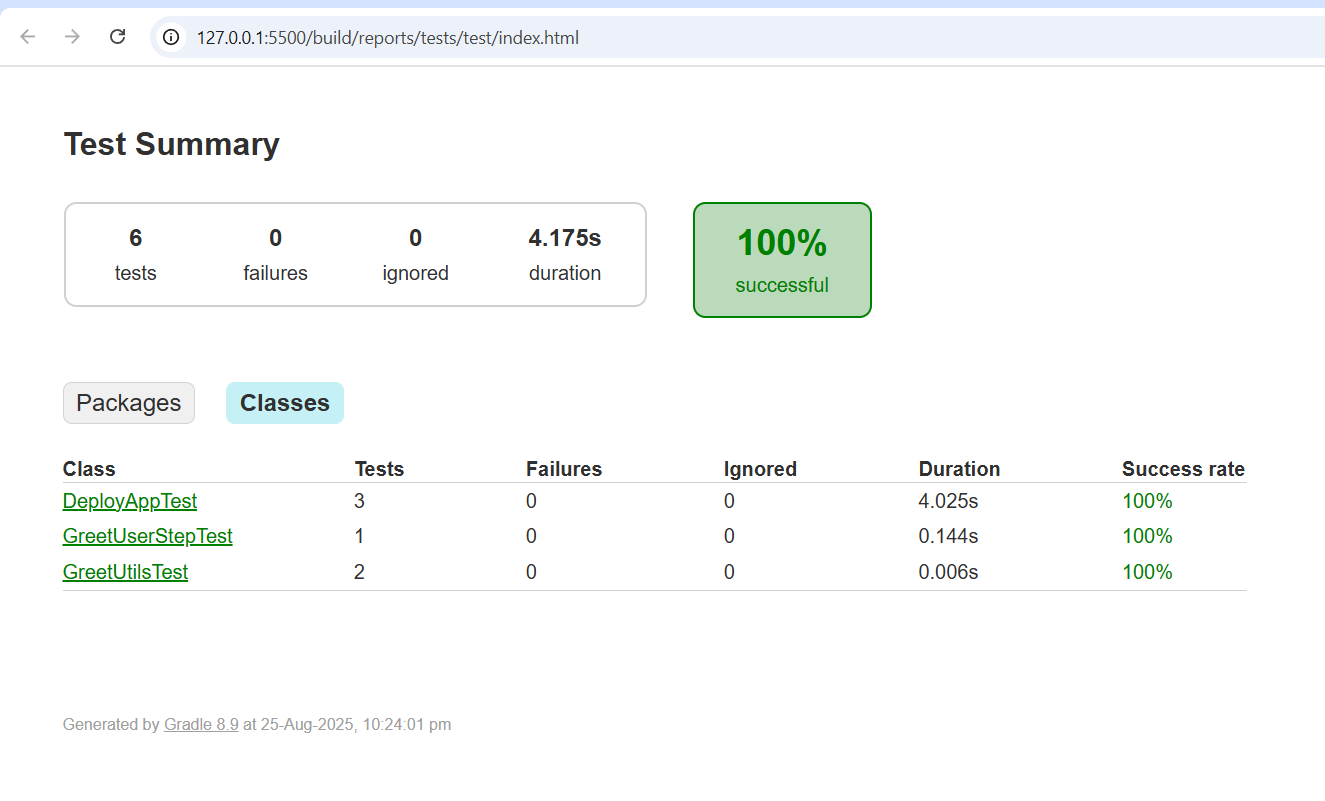
1. Don’t write everything in /vars → push complex logic to /src.
2. Use JPU mocks only where Jenkins DSL is unavoidable (sh, checkout, etc.).
3. Business logic = plain Groovy = no mocking.
4. Keep /resources under test to prevent silent config/template errors.
5. CI/CD enforces tests: break the build if shared library unit tests fail.

✅ With this approach, you:

* Catch pipeline errors *before Jenkins runtime*.
* Keep business logic testable without JPU.
* Have a repeatable POC → production adoption path.

Would you like me to now draft a sample “end-to-end POC test repo” (with /vars, /src, /resources, and matching JPU tests), so you can just clone + demo it at work?

ChatGPT can make mistakes. Check important info. See Cookie Preferences.

All tests:  


Refactor for DI (preferred) OR metaClass hack what is DI and is it related jpu or what also what refractoring means do we have to tweek the original code

**ChatGPT said:**

Great questions 👍 Let me break this down clearly without buzzwords.

**1. What is DI (Dependency Injection)?**

* **Definition (simple)**: Instead of a class creating/controlling everything inside itself, it *receives* the things it depends on from the outside.
* **Why**: Makes code easier to test, reuse, and extend.

**Example without DI (hard to test):**

class DeployApp {

def deploy() {

def docker = new DockerClient() // 👈 hard-coded, cannot swap for a fake

docker.run("my-app")

}

}

**With DI (easy to test):**

class DeployApp {

DockerClient docker // 👈 injected from outside

def deploy() {

docker.run("my-app")

}

}

Now in **JenkinsPipelineUnit (JPU) tests**, you can pass a fake DockerClient or mock instead of using the real one.

**2. Is DI related to JPU?**

* Not originally. DI is a **general software design principle**.
* But **JPU struggles with direct constructors (new) and static calls**, because it can’t intercept them.
* So in Jenkins Shared Libraries, we apply DI so that JPU can replace those dependencies with mocks or stubs in unit tests.

➡️ So yes, DI becomes **critical for JPU testing**, but it’s not invented by JPU.

**3. What does refactoring mean here?**

* Refactoring = **changing the internal structure of the code without changing what it does**.
* You don’t change the *feature/output*, just the *design/structure* to make it more testable.

**Example (vars/deployApp.groovy before refactor):**

def call(String image) {

def docker = new DockerClient() // tightly coupled

docker.run(image)

}

**After refactor for DI (vars/deployApp.groovy):**

def call(String image, dockerClient = new DockerClient()) { // 👈 allow injection

dockerClient.run(image)

}

Now in production pipeline:

deployApp("my-app") // uses real DockerClient

In JPU test:

def fakeDocker = [ run: { img -> println "Pretend running $img" } ]

script.call("my-app", fakeDocker) // inject fake dependency

**4. So, do we have to tweak the original code?**

* **Yes, but carefully**.
* Jenkins Shared Library steps in /vars or helpers in /src may need slight refactoring to support DI.
* These tweaks don’t change the pipeline’s behavior in Jenkins, but they make the steps *testable locally with JPU*.

✅ **Summary in one line:**

* **DI** = inject dependencies instead of creating them internally.
* It’s **not unique to JPU**, but JPU makes it necessary because it cannot mock constructors/static methods.
* **Refactoring** = slight tweaks to your shared library code (e.g., allow passing dependencies) so you can test with JPU **without breaking production pipelines**.

Would you like me to create a **before/after refactor POC doc** (showing /vars, /src, and test examples) so you can present how DI makes JPU work in real-world Jenkins Shared Libraries?

**You said:**

yes

**ChatGPT said:**

Perfect 👍 Let’s put this into a **POC-style doc** you can present to your team.  
I’ll structure it like a lightweight standard for **JenkinsPipelineUnit (JPU) + Shared Libraries testing**, with **before/after DI refactor examples**.

**📄 POC: Dependency Injection & Testing Jenkins Shared Libraries with JPU**

**1. Why This Document?**

JenkinsPipelineUnit (JPU) is the de-facto framework for unit testing Jenkins Shared Libraries (/vars, /src, /resources).  
However, JPU **cannot mock constructors (new) or static calls reliably**.  
To make our shared libraries testable, we apply **Dependency Injection (DI)** + minor **refactoring**.

**2. Critical Setup for JPU**

* **Local Setup**
  + build.gradle with dependencies:
  + testImplementation 'junit:junit:4.13.2'
  + testImplementation 'com.lesfurets:jenkins-pipeline-unit:1.16'
  + testImplementation 'org.assertj:assertj-core:3.25.3'
  + Unit test runner: JUnit
  + Test sources: src/test/groovy
* **Jenkins Setup**
  + Same repo (Shared Library) loaded into Jenkins
  + JPU tests run in CI before publishing library
  + Can run in Jenkinsfile with:
  + stage('Unit Test') {
  + steps {
  + sh './gradlew test'
  + }
  + }

**3. Problem: Code Without DI**

**Example: /vars/deployApp.groovy**

def call(String image) {

def docker = new DockerClient() // hard-coded 👎

docker.run(image)

}

* **Why it fails in JPU?**
  + JPU cannot intercept new DockerClient().
  + Unit test would try to create a real DockerClient, which may not exist locally.

**4. Solution: Refactor for DI (Preferred)**

**Refactored /vars/deployApp.groovy**

def call(String image, dockerClient = new DockerClient()) {

dockerClient.run(image)

}

**Pipeline usage (no change):**

deployApp("my-service:1.0") // still works in Jenkins

**JPU Test Example (src/test/groovy/DeployAppTest.groovy)**

class DeployAppTest extends BasePipelineTest {

@Test

void testWithMock() {

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

def fakeDocker = [ run: { img -> "Fake running $img" } ]

def result = script.call("test-app", fakeDocker)

assertEquals("Fake running test-app", result)

}

}

✅ Now tests run locally, with no dependency on real Docker.

**5. Alternative: MetaClass Hack (Less Clean)**

Instead of DI, we can patch the class using Groovy metaClass:

DockerClient.metaClass.constructor = { -> [ run: { img -> "Mocked $img" } ] }

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

assertEquals("Mocked my-app", script.call("my-app"))

⚠️ This works but:

* Fragile
* Harder to maintain
* Can break with future Groovy versions

**6. DI in /src/ Classes**

Example: /src/org/example/Deployer.groovy

**Before:**

class Deployer {

def deploy(String image) {

def docker = new DockerClient()

docker.run(image)

}

}

**After (refactored for DI):**

class Deployer {

DockerClient docker

Deployer(docker = new DockerClient()) { this.docker = docker }

def deploy(String image) {

docker.run(image)

}

}

**Test:**

class DeployerTest {

@Test

void testDeployerWithMock() {

def fakeDocker = [ run: { img -> "MockRun $img" } ]

def deployer = new Deployer(fakeDocker)

assertEquals("MockRun my-app", deployer.deploy("my-app"))

}

}

**7. /resources Testing**

* /resources usually contains Groovy templates, JSON, YAML, or shell scripts.
* You don’t need DI here; just load files with libraryResource.
* In JPU, you mock them with:

helper.registerSharedLibraryResource("myconfig.yml", "env: test\nversion: 1.0")

**8. Summary: Best Practices**

* ✅ Always prefer **Dependency Injection (DI)** when writing /vars or /src logic.
* ✅ Keep /vars scripts thin; push logic into /src classes (testable).
* ✅ Use **JPU + JUnit** locally before pushing library updates.
* ✅ Use **Mock objects** or fakes instead of hitting real tools (Docker, Git, etc.).
* ⚠️ Avoid direct constructors (new) or static utility calls in pipeline code.
* ⚠️ Avoid overusing metaClass (last resort).

📌 **One-liner takeaway for presentation:**

JPU testing works best when our shared library code is written with **Dependency Injection** in mind.  
Refactor to accept dependencies instead of hardcoding them — this unlocks clean, reliable unit tests for /vars, /src, and even /resources.

Would you like me to also add a **comparison table (With DI vs Without DI)** so you can quickly show your team the benefits in a presentation slide?

**You said:**

comparison table (With DI vs Without DI) so you can quickly show your team the benefits in a presentation slide yes give me

**ChatGPT said:**

Perfect 👍 — this is a great idea because many teams get confused on *why* we should tweak (refactor) shared library code for **Dependency Injection (DI)** instead of sticking with metaClass hacks.  
Here’s a **comparison table (DI vs non-DI)** that you can directly use in your **POC presentation / team discussion**:

**📊 Comparison: With DI vs Without DI in JenkinsPipelineUnit (JPU) Testing**

| **Aspect** | **With DI (Dependency Injection)** | **Without DI (metaClass Hack / Hard Dependency)** |
| --- | --- | --- |
| **Code Change Required** | Yes – small refactor: pass dependencies (e.g., utils, http client) as parameters or assign them to a field. | No – keep existing code as-is, but patch/mutate with Groovy metaClass. |
| **Testability** | ✅ Easy to test. You can pass in mocks, stubs, or real implementations. | ⚠️ Harder. Must override methods at runtime using metaClass. Fragile if class names/methods change. |
| **Reliability of Tests** | ✅ Stable across versions of Groovy/Jenkins. | ❌ Tests can break if Jenkins core / JPU internals change. |
| **Readability** | ✅ Code clearly shows what dependencies exist. | ❌ Tests look “magical”; developers may not understand how metaClass mocking works. |
| **Maintainability** | ✅ Long-term friendly; easier for new team members. | ❌ Technical debt accumulates; debugging becomes painful. |
| **Production Code Impact** | Minimal – only adds optional injection points. Logic doesn’t change. | None in production, but test code becomes hacky. |
| **Industry Best Practice** | ✅ Widely recommended in CI/CD shared library projects. | ❌ Usually discouraged except for very quick POCs or legacy code. |
| **Learning Curve** | Slightly higher – developers need to understand DI pattern. | Lower initially, but long-term complexity grows. |
| **Use Cases** | Production-grade shared libraries, critical pipelines, team adoption. | Quick proof-of-concept, testing legacy libs without refactor. |

**📝 Example Contrast**

**Without DI (Hard-coded dependency):**

// vars/deployApp.groovy

def call(String appName) {

def utils = new Utils() // hardcoded

echo utils.deploy(appName)

}

**With DI (Refactored for Dependency Injection):**

// vars/deployApp.groovy

def call(String appName, utils = new Utils()) {

echo utils.deploy(appName) // can inject mock or real Utils

}

**Unit Test With DI:**

@Test

void testWithDI() {

def mockUtils = [ deploy: { name -> "Mock Deploy $name" } ] as Utils

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

def result = script.call("MyApp", mockUtils)

assert result == "Mock Deploy MyApp"

}

✅ **Summary for your POC**:

* **DI = scalable, clean, stable** (recommended).
* **metaClass = quick hack** (only for legacy libs you can’t touch).
* For your team, show this comparison and recommend **adopting DI for all new shared libraries**.

Would you like me to **format this into a slide-deck style (bullet + visual)** so you can directly drop it into your POC presentation, instead of just a text table?

**You said:**

import org.junit.Before import org.junit.Test import static org.junit.Assert.\* import com.lesfurets.jenkins.unit.BasePipelineTest import org.example.utils.DeployUtils class DeployAppTest extends BasePipelineTest { def deployAppScript @Before void setUp() throws Exception { super.setUp() // Load the shared library step under test deployAppScript = loadScript("vars/deployApp.groovy") // Mock echo to print to console helper.registerAllowedMethod("echo", [String]) { msg -> println msg } // Mock error step to throw RuntimeException helper.registerAllowedMethod("error", [String]) { msg -> throw new RuntimeException(msg) } } @Test void testSuccessfulDeployment() { // Inject a mock DeployUtils def mockDeployUtils = [deploy: { app, version -> "SUCCESS" }] def result = deployAppScript.call("myApp", "1.0.0", "dev", mockDeployUtils) assertEquals("SUCCESS", result) } @Test void testDeploymentFailure() { // Inject a mock DeployUtils that returns FAILED def mockDeployUtils = [deploy: { app, version -> "FAILED" }] boolean exceptionThrown = false try { deployAppScript.call("myApp", "1.0.0", "dev", mockDeployUtils) } catch (RuntimeException e) { exceptionThrown = e.message.contains("Deployment failed") } assertTrue("Expected RuntimeException for failed deployment", exceptionThrown) } @Test void testMissingParams() { boolean exceptionThrown = false try { deployAppScript.call(null, "1.0.0", "dev") } catch (RuntimeException e) { exceptionThrown = e.message.contains("Missing required parameters") exceptionThrown = true } assertTrue("Expected RuntimeException for missing parameters", exceptionThrown) } } and var is import org.example.utils.DeployUtils def call(String appName, String version, String targetEnv, deployUtils = null) { if (!appName || !version || !targetEnv) { error "Missing required parameters: appName, version, targetEnv" } // Use injected deployUtils or create a new one def utils = deployUtils ?: new DeployUtils(this, targetEnv) echo "Starting deployment of ${appName}:${version} to ${targetEnv}" def result = utils.deploy(appName, version) if (result != "SUCCESS") { error "Deployment failed for ${appName}:${version} on ${targetEnv}" } echo "Deployment of ${appName}:${version} to ${targetEnv} completed successfully" return result } explain me very clen way so non coding person understands how this refractor works? when we need to refractor while we need to mock or everytime where exatly i can know to refractor at what instance

**ChatGPT said:**

Perfect question 👍 — you’re getting to the *heart* of why we even bother with **refactoring for DI** in Jenkins Shared Library unit tests. Let me explain in a very **clean, non-coder way** so you can also present this clearly to your team.

**1. What’s happening in your code today?**

You have two files:

* **vars/deployApp.groovy (the pipeline step)** → this is what Jenkins calls inside a pipeline.
* **src/org/example/utils/DeployUtils.groovy (a helper class)** → this does the actual deployment logic.

Normally, deployApp would *create its own DeployUtils object* (using new DeployUtils()), and then use it.  
That means your test has **no control** over what DeployUtils does inside.

**2. The Problem (without refactoring)**

* If DeployUtils does something complicated (like connecting to Kubernetes, AWS, etc.), your unit test will try to run it → 💥 it fails, because your test environment doesn’t have those systems.
* JenkinsPipelineUnit (JPU) cannot easily “fake” (mock) new object creation (new DeployUtils()), because the constructor is hard-wired.
* Result → your test becomes **brittle, slow, or impossible to run locally**.

**3. The Refactor (Dependency Injection — DI)**

Instead of deployApp always creating its own DeployUtils,  
we **give it the option** to accept a ready-made one, like this:

def call(String appName, String version, String targetEnv, deployUtils = null) {

// If none provided, create the real one

def utils = deployUtils ?: new DeployUtils(this, targetEnv)

...

}

Now in **real Jenkins pipelines**, you don’t pass anything → so it creates the real DeployUtils and works normally. ✅

But in **tests**, you pass a fake one (mockDeployUtils) →  
this allows you to control its behavior (success, failure, etc.) and test **your pipeline logic**, without worrying about real AWS/K8s. ✅

**4. Analogy (for non-coders)**

Imagine you’re testing a car 🚗:

* Without DI → every time you test, the car *always* comes with its real engine. If the engine is missing fuel or has a fault, your test fails — even though you just wanted to check if the *dashboard lights* turn on.
* With DI → you can plug in a fake engine (a simulator) just for testing. Now you can test the dashboard logic safely, without needing real fuel or exhaust pipes.

**5. When do you need to refactor?**

You **only refactor for DI when:**

* Your step (vars/...) creates a **new object (new SomeClass)** inside it.
* That object does real work (network calls, API calls, filesystem).
* You want your test to control the outcome of that work.

👉 If your function only uses Jenkins steps like echo, sh, error → no need for DI, you just mock those steps in your test.  
👉 If your function depends on helpers (DeployUtils, GitUtils, etc.) → refactor to allow DI.

**6. When mocking is enough (no refactor needed)?**

* If it’s a Jenkins step (sh, git, error, echo, etc.) → JPU lets you mock these easily.
* If it’s a pure Groovy method already passed in as a parameter → also mock directly.
* **You only need DI refactor when new is being used inside your pipeline code.**

✅ **Summary (non-boring, POC-style message to team):**

* **Without DI** → our tests break because they try to run *real AWS/K8s code*.
* **With DI** → we can inject fake helpers, so our tests check *pipeline logic only*, not external systems.
* We **refactor only where new is used**, not everywhere.
* This keeps our pipeline library **testable locally** and **safe in Jenkins CI**.

👉 Now my question to you:  
Do you want me to make a **decision tree / checklist** like:  
*"If you see this in code → refactor; if you see that → just mock"* so your team knows when DI is required?

**You said:**

o you want me to make a decision tree / checklist like: "If you see this in code → refactor; if you see that → just mock" so your team knows when DI is required? yes give me that

**ChatGPT said:**

Perfect 👍 Let’s build you a **decision tree / checklist** that’s simple enough for even non-coders in your team to follow.

**✅ Decision Tree: When to Refactor (DI) vs When to Just Mock in JenkinsPipelineUnit (JPU)**

**1. Look at your vars/ or src/ code**

* Does the code **create new objects directly**?  
  Example:
* def utils = new DeployUtils(this, targetEnv)

👉 **Refactor with DI (Dependency Injection)**

* Does the code **use global Jenkins steps only** (e.g., sh, echo, git, error)?  
  Example:
* sh "mvn clean install"
* echo "Build done"

👉 **No refactor needed, just mock those steps in your test**

**2. Ask: “Do I control what gets injected?”**

* If yes → ✅ Refactor to **accept it as a parameter** so you can inject a fake/mock in tests.
* If no → ❌ You’re stuck with the hardcoded class/object → unit testing becomes unreliable.

**3. Rule of Thumb Checklist 📝**

* **Constructor calls (new Something()) in your code?** → Refactor with DI.
* **Static utility methods (SomeClass.someStaticMethod())** → Prefer refactor (wrap it or inject it).
* **Pipeline steps (sh, echo, error, checkout)** → Mock them in JPU, no refactor needed.
* **Groovy metaClass hacking** → Use **only if** refactor is impossible (short-term fix).

**4. Example in Practice**

**❌ Without DI (Hard to test)**

def call(String app, String version, String env) {

def utils = new DeployUtils(this, env) // Hardcoded constructor

utils.deploy(app, version)

}

* Test is stuck with a real DeployUtils.
* You need JPU to hack it with metaClass, which is brittle.

**✅ With DI (Refactored)**

def call(String app, String version, String env, deployUtils = null) {

def utils = deployUtils ?: new DeployUtils(this, env) // DI

utils.deploy(app, version)

}

* In tests, you can pass a **fake object**:
* def mockUtils = [deploy: { a, v -> "SUCCESS" }]
* deployAppScript.call("app", "1.0", "dev", mockUtils)
* No brittle hacks, fully testable.

**5. Simple Team-Friendly Mnemonic 🧠**

👉 **“If you see new, think DI.”**  
👉 **“If it’s a pipeline step, just mock.”**

⚡ So in your example (DeployUtils), we **had to refactor** because the vars/deployApp.groovy created a new object with new DeployUtils(...). Without that, your test would fail or force ugly hacks.

Do you want me to also make a **slide-style one-pager checklist** (like a presentation diagram with arrows: *If new → Refactor, If step → Mock*) so you can directly present it to your team?