**TDD FileLoader v1.4**

Using mocking in the test

**Unit Test**

import unittest

from app.file\_loader import FileLoader

class TestFileLoader(unittest.TestCase):

def test\_load\_all\_of\_file\_using\_inbuilt\_files\_type\_as\_lambda(self):

# Arrange

file\_to\_load = "sample.txt"

cut = FileLoader(file\_to\_load)

expected\_bytes\_read = 12

# Calculate expected number of characters

with open(file\_to\_load, encoding="utf-8") as f:

expected\_bytes\_read = sum(len(line) for line in f.readlines())

# Act

bytes\_read = cut.load\_file\_with\_func(lambda fname: open(fname,

encoding="utf-8").readlines())

# Assert

self.assertEqual(expected\_bytes\_read, bytes\_read)

def test\_load\_all\_of\_file\_via\_stub(self):

""" Use a hardcoded stub to simulate reading two lines of text

Benefit - no dependency on actual files or filesystem

- portable test

- FileLoader is more flexible and decoupled allowing

file loading mechanism to be injected

"""

# arrange

file\_to\_load = ""

cut = FileLoader(file\_to\_load)

expected\_bytes\_read = 10

# act

bytes\_read = cut.load\_file\_with\_func(lambda fname: ["Hello", "world"])

# assert

self.assertEqual(expected\_bytes\_read, bytes\_read)

def test\_load\_all\_of\_file\_using\_mock(self):

# Arrange

file\_to\_load = "c:/tmp/KeyboardHandler.txt"

cut = FileLoader(file\_to\_load)

# Simulate file content

pretend\_file\_content = ["Hello", "world"]

expected\_bytes\_read = 10

# Create a mock callable for reading files

mock\_file\_reader = Mock()

mock\_file\_reader.return\_value = pretend\_file\_content

# Act

# bytes\_read = cut.load\_file\_with\_func(lambda fname: mock\_file\_reader(fname))

bytes\_read = cut.load\_file\_with\_func(lambda fname: mock\_file\_reader("XYZ"))

# Assert

self.assertEqual(expected\_bytes\_read, bytes\_read)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**The CUT FileLoader**

"""

FileLoader Module

This module defines the FileLoader class which is responsible for reading a text file

and calculating the total size (in characters) of its contents. It also supports

dependency injection through the `load\_file\_with\_func` method, allowing testability

without relying on actual file I/O operations.

This is useful in unit testing scenarios where file system access should be avoided.

"""

class FileLoader:

def \_\_init\_\_(self, file\_to\_load):

self.file\_to\_load = file\_to\_load

self.lines = []

def load\_file(self, fname):

"""

Loads a file from disk and reads its contents line by line.

Falls back to an empty list if the file cannot be read.

"""

try:

with open(fname, encoding='utf-8') as f:

self.lines = f.readlines()

except IOError:

self.lines = []

return self.\_calculate\_file\_size()

def get\_lines(self):

"""Returns the list of lines read from the file."""

return self.lines

def load\_file\_with\_func(self, func):

"""

Accepts a file loading function to inject lines, used primarily for testing.

This avoids direct I/O operations and makes the method more testable by passing

a mock or simulated version of file loading logic.

"""

self.lines = func(self.file\_to\_load)

return self.\_calculate\_file\_size()

def \_calculate\_file\_size(self):

return sum(len(line) for line in self.lines)