Complete TDD Implementation Guide - Corrected Version

Step 1: Understanding TDD Principles

TDD Cycle (Red-Green-Refactor):

1. **Red**: Write a failing test first
2. **Green**: Write minimal code to make the test pass
3. **Refactor**: Improve code while keeping tests green

Step 2: Project Setup

Create project structure:

markdown Copy code

```
jira_analyzer_project/ ├── jira_analyzer/ │ ├── __init__.py │ └── analyzer.py ├── test/ │ ├── __init__.py │ └── test_analyzer.py ├── pytest.ini ├── conftest.py ├── requirements.txt └── requirements-test.txt
```

Install dependencies:

bash Copy code

```
pip install -r requirements.txt pip install -r requirements-test.txt
```

Step 3: Core Files Setup

`requirements.txt`:

txt Copy code

```
jira==3.5.0
```

`requirements-test.txt`:

txt Copy code

```
pytest==7.4.0 pytest-mock==3.11.1 pytest-cov==4.1.0
```

`pytest.ini`:

ini Copy code

```
[tool:pytest] testpaths = test python_files = test_*.py python_classes = Test* python_functions = test_* addopts = -v --cov=jira_analyzer --cov-report=html --
```

```
cov-report=term-missing [tool:pytest.ini_options] markers = unit: marks tests as unit tests (fast, isolated) integration: marks tests as integration tests (slower, multiple components) slow: marks tests as slow tests (performance, large datasets)
```

`conftest.py`:

python Copy code

```
import pytest @pytest.fixture(autouse=True) def setup_test_environment(): """Setup that runs before each test""" yield
```

`jira_analyzer/__init__.py`:

python Copy code

```
from .analyzer import JiraStatusAnalyzer __version__ = "1.0.0"
```

`test/__init__.py`:

python Copy code

```
# Empty file - makes tests a package
```

Step 4: Main Application Code

`jira_analyzer/analyzer.py`:

(Use the refactored code from the original Step 3 - no changes to the class structure)

Step 5: Basic Test Suite

`test/test_analyzer.py`:

python Copy code

```
import sys import os import pytest from datetime import datetime from unittest.mock import Mock, patch # Add parent directory to path for imports sys.path.insert(0, os.path.dirname(os.path.dirname(os.path.abspath(__file__)))) from jira_analyzer import JiraStatusAnalyzer class TestJiraStatusAnalyzer: def setup_method(self): """Setup for each test""" self.analyzer = JiraStatusAnalyzer("https://test-jira.com", "test-token") @pytest.mark.unit def test_analyzer_initialization(self): """Test that analyzer initializes correctly""" assert self.analyzer.jira_url == "https://test-jira.com" assert self.analyzer.api_token == "test-token" assert self.analyzer.jira is None @pytest.mark.unit @patch('jira_analyzer.analyzer.JIRA') def test_connect_to_jira_success(self, mock_jira): """Test successful Jira
```

```
connection""" mock_jira_instance = Mock() mock_jira.return_value =
mock_jira_instance result = self.analyzer.connect_to_jira() assert result is True
assert self.analyzer.jira == mock_jira_instance mock_jira.assert_called_once()
@pytest.mark.unit @patch('jira_analyzer.analyzer.JIRA') def
test_connect_to_jira_failure(self, mock_jira): """Test Jira connection failure"""
mock_jira.side_effect = Exception("Connection failed") result =
self.analyzer.connect_to_jira() assert result is False assert self.analyzer.jira
is None @pytest.mark.unit def test_calculate_days_difference(self): """Test days
calculation between dates""" start_date = datetime(2024, 1, 1) end_date =
datetime(2024, 1, 11) result =
self.analyzer.calculate_days_difference(start_date, end_date) assert result == 10
@pytest.mark.unit def test_calculate_days_difference_none_values(self): """Test
days calculation with None values""" assert
self.analyzer.calculate_days_difference(None, None) is None assert
self.analyzer.calculate_days_difference(datetime.now(), None) is None
```

Step 6: TDD Workflow - Learning the Red-Green-Refactor Cycle

Example: Adding Issue Key Validation (Complete TDD Example)

Phase 1: RED - Write a Failing Test First

Add this test to `test/test_analyzer.py`:

python Copy code

```
@pytest.mark.unit def test_validate_issue_key_format_valid(self): """Test valid
issue key format - RED PHASE""" # This test will FAIL because validate_issue_key
doesn't exist yet assert self.analyzer.validate_issue_key("PROJ-123") is True
assert self.analyzer.validate_issue_key("ABC-999") is True assert
self.analyzer.validate_issue_key("ISDOP-1234") is True @pytest.mark.unit def
test_validate_issue_key_format_invalid(self): """Test invalid issue key format -
RED PHASE""" # This test will FAIL because validate_issue_key doesn't exist yet
assert self.analyzer.validate_issue_key("invalid") is False assert
self.analyzer.validate_issue_key("123") is False assert
self.analyzer.validate_issue_key("") is False assert
self.analyzer.validate_issue_key("PROJ") is False assert
self.analyzer.validate_issue_key("PROJ-") is False
```

Run the test (it should FAIL - RED):

bash Copy code

```
pytest
test/test_analyzer.py::TestJiraStatusAnalyzer::test_validate_issue_key_format_valid
```

```
-v
```

Expected output: `AttributeError: 'JiraStatusAnalyzer' object has no attribute 'validate_issue_key'`

Phase 2: GREEN - Write Minimal Code to Make Test Pass

Add this method to `jira_analyzer/analyzer.py` in the `JiraStatusAnalyzer` class:

python Copy code

```python
def validate_issue_key(self, issue_key): """Validate issue key format (PROJECT-NUMBER) - GREEN PHASE""" if not issue_key: return False import re pattern = r'^[A-Z]+-\d+$' return bool(re.match(pattern, issue_key))
```

Run the test again (it should PASS - GREEN):

bash Copy code

```bash
pytest test/test_analyzer.py::TestJiraStatusAnalyzer::test_validate_issue_key_format_valid -v pytest test/test_analyzer.py::TestJiraStatusAnalyzer::test_validate_issue_key_format_invali -v
```

Expected output: All tests should pass ✅

Phase 3: REFACTOR - Improve Code While Keeping Tests Green

Improve the implementation in `jira_analyzer/analyzer.py`:

python Copy code

```python
def validate_issue_key(self, issue_key): """ Validate issue key format (PROJECT-NUMBER) - REFACTOR PHASE Args: issue_key (str): The issue key to validate Returns: bool: True if valid, False otherwise Examples: >>> analyzer.validate_issue_key("PROJ-123") True >>> analyzer.validate_issue_key("invalid") False """ if not issue_key or not isinstance(issue_key, str): return False import re # Match: 1+ uppercase letters, hyphen, 1+ digits pattern = r'^[A-Z]{2,}-\d+$' # Improved: at least 2 letters for project return bool(re.match(pattern, issue_key.strip()))
```

Run all tests to ensure refactoring didn't break anything:

bash Copy code

```
pytest test/test_analyzer.py -v
```

All tests should still pass ✅

Step 7: Advanced TDD - Integration Test Example

RED Phase: Write Integration Test

Add this test to `test/test_analyzer.py`:

python Copy code

```
@pytest.mark.integration def test_parse_and_validate_csv_row_integration(self): """Integration test - RED PHASE: Parse row and validate issue key""" # This will fail because we haven't integrated validation into parsing yet # Valid case issue_key, error = self.analyzer.parse_and_validate_csv_row(["PROJ-123", "Summary"], 1) assert issue_key == "PROJ-123" assert error is None # Invalid case issue_key, error = self.analyzer.parse_and_validate_csv_row(["invalid-key", "Summary"], 1) assert issue_key is None assert "Invalid issue key format" in error
```

Run test (should FAIL - RED):

bash Copy code

```
pytest test/test_analyzer.py::TestJiraStatusAnalyzer::test_parse_and_validate_csv_row_integ -v
```

GREEN Phase: Implement Integration

Add this method to `jira_analyzer/analyzer.py`:

python Copy code

```
def parse_and_validate_csv_row(self, row, row_number): """ Parse CSV row and validate issue key format - GREEN PHASE """ # First parse the row issue_key, error = self.parse_csv_row(row, row_number) if error: return None, error # Then validate the issue key format if not self.validate_issue_key(issue_key): return None, f"Row {row_number}: Invalid issue key format '{issue_key}'" return issue_key, None
```

Run test (should PASS - GREEN):

bash Copy code

```
pytest
test/test_analyzer.py::TestJiraStatusAnalyzer::test_parse_and_validate_csv_row_integ
-v
```

Step 8: TDD Best Practices Commands

Run Different Test Categories:

bash Copy code

```
# Run only unit tests (fast) pytest -m unit # Run only integration tests pytest -
m integration # Run all tests with coverage pytest --cov=jira_analyzer # Run
specific test file pytest test/test_analyzer.py -v # Run tests matching pattern
pytest -k "validate" -v
```

TDD Development Workflow:

bash Copy code

```
# 1. RED: Write failing test pytest
test/test_analyzer.py::TestJiraStatusAnalyzer::test_new_feature -v # 2. GREEN:
Write minimal code, run test pytest
test/test_analyzer.py::TestJiraStatusAnalyzer::test_new_feature -v # 3. REFACTOR:
Improve code, run all tests pytest test/test_analyzer.py -v # 4. Repeat cycle for
next feature
```

Step 9: Proving Your Tests Work (Mutation Testing)

Deliberately Break Code to Verify Tests Catch Issues:

In `jira_analyzer/analyzer.py` , temporarily break the `calculate_days_difference` method:

python Copy code

```
def calculate_days_difference(self, start_date, end_date): """BROKEN VERSION -
for testing""" if start_date and end_date: return (end_date - start_date).days +
999 # ← DELIBERATELY WRONG return None
```

Run tests (they should FAIL, proving they work):

bash Copy code

```
pytest
test/test_analyzer.py::TestJiraStatusAnalyzer::test_calculate_days_difference -v
```

Revert the change and tests should pass again.

Step 10: Complete TDD Cycle Example

Here's a complete example you can practice with:

1. RED - Add New Feature Test

python Copy code

```
@pytest.mark.unit def test_format_date_string(self): """Test date formatting -
RED PHASE""" date_obj = datetime(2024, 1, 15) result =
self.analyzer.format_date_string(date_obj) assert result == "2024-01-15" # Test
None handling result = self.analyzer.format_date_string(None) assert result == ""
```

2. Run Test (RED)

bash Copy code

```
pytest test/test_analyzer.py::TestJiraStatusAnalyzer::test_format_date_string -v
# Should fail with AttributeError
```

3. GREEN - Implement Method

python Copy code

```
def format_date_string(self, date_obj): """Format date object to string - GREEN
PHASE""" if date_obj is None: return "" return date_obj.strftime('%Y-%m-%d')
```

4. Run Test (GREEN)

bash Copy code

```
pytest test/test_analyzer.py::TestJiraStatusAnalyzer::test_format_date_string -v
# Should pass
```

5. REFACTOR - Improve Implementation

python Copy code

```
def format_date_string(self, date_obj, format_string='%Y-%m-%d'): """ Format date
object to string - REFACTOR PHASE Args: date_obj: datetime object or None
```

```
format_string: strftime format string Returns: str: Formatted date string or
empty string if None """ if date_obj is None: return "" try: return
date_obj.strftime(format_string) except (AttributeError, ValueError): return ""
```

6. Run All Tests (Should Still Pass)

bash Copy code

```
pytest test/test_analyzer.py -v
```

Summary: Your TDD Learning Checklist

✅ **RED**: Always write the test first (it should fail)

✅ **GREEN**: Write minimal code to make the test pass

✅ **REFACTOR**: Improve code while keeping tests green

✅ **Verify**: Break code intentionally to prove tests work

✅ **Repeat**: Continue cycle for each new feature

✅ **Categories**: Use `@pytest.mark.unit` , `@pytest.mark.integration`

✅ **Run Often**: `pytest -v` after every change

This procedure gives you a complete, working TDD environment where you can practice the Red-Green-Refactor cycle safely!