# PYNOSE: A Test Smell Detector For Python

https://arxiv.org/pdf/2108.04639.pdf

*Tongjie Wang - University of California, Irvine*
*Yaroslav Golubev - JetBrains Research*
*Oleg Smirnov - JetBrains Research*
*Jiawei Li - University of California, Irvine*
*Timofey Bryksin - JetBrains Research*
*Iftekhar Ahmed - University of California, Irvine*

# What are code smells?

- Smells are certain structures in the **code** that indicate **violation** of fundamental design principles.

- **Subjective**, and varies by language, developer, and development methodology.

- They are certainly **not bugs**, but increase the risk of bugs or failures in the future.

# Unit testing

- The goal is to isolate each part of the program and show that the individual parts are **correct**.

- It provides a strict, **written contract** that the piece of code must satisfy mostly using **validations**.

- **Safety net** to refactor code with confidence.

- Tests are your **documentation** which talks about the importance in maintainability of software.

# Statically typed languages

- It is very popular among development of **backend systems**.

- Main reason being **compile time** safety leading to less bugs and easier maintenance.

- Expressing **business domains using types** and building business logic around them.

- Majority of **test smells research** has been focused around these languages.

# Python

- **Dynamically** typed language.

- Hugely popular and adapted in **data science and machine learning** projects.

- This paper is mostly focused on studying test smells in **Python projects**.

# Goal of the paper

- Identifying **test smells** from well established open source **python projects**.

- Providing a tool (**PYNOSE**) for the test smell detection during development stages and also **offline** analysis.

- Also an **empirical** study of test smell **pervasiveness** by applying the PYNOSE tool on matured python projects.

# Systematic mapping study

- Keyword search of **"test smell" or "test smells"** in digital libraries like ACM, IEEE and Scopus.

- Restrict to **digital** publications about **software** engineering methodologies, test **smell detection** and **refactoring.**

- There could be potential articles left out due to the above criteria and hence single iteration of **backward snowballing** is conducted on **left out papers**.

- Reliability of selected publications was **manually evaluated**.

# Identifying test smells

- Focus was only on **Unittest** testing framework that is included in the Python Standard Library.

- Some of them are **only valid** for statically typed languages like **Java** or **Scala**.

- After the above process **17 smells** were **shortlisted**.

- Shortlisted test smells are **implemented** in Python.

# Assertion Roulette

A test case that contains more than one assertion statement without an explanation/message.

Calculator.py | test_Calculator.py | case.py | Pipfile

```python
from unittest import TestCase

from Calculator import Calculator


class TestCalculator(TestCase):

    def setUp(self):
        self.calculator = Calculator()

    def test_add(self):
        expected = 30
        a = 10
        b = 20
        result1 = self.calculator.add(a, b)
        result2 = self.calculator.add(b, a)
        self.assertEqual(expected, result1)
        self.assertEqual(expected, result2)
```

PyNose

**Smells Detected by PyNose**

**test_Calculator.py**

**TestCalculator**

- AssertionRoulette

# Conditional Test Logic

A test case that contains one or more control statements (i.e., if, for, while)

Calculator.py | test_Calculator.py | case.py | Pipfile

```python
from unittest import TestCase

from Calculator import Calculator


class TestCalculator(TestCase):

    def setUp(self):
        self.calculator = Calculator()

    def test_add(self):
        additional_parameter = 1
        a = 10
        b = 20
        expected = 30
        if additional_parameter == 1:
            result = self.calculator.add(a, b)
            self.assertEqual(expected, result)
```

PyNose

**Smells Detected by PyNose**

**test_Calculator.py**

**TestCalculator**

- ConditionalTestLogic

# Constructor Initialization

A test case that contains more than one assertion statement without an explanation/message.

Calculator.py | test_Calculator.py | case.py | Pipfile

```python
from unittest import TestCase

from Calculator import Calculator


class TestCalculator(TestCase):

    def __init__(self):
        super().__init__()
        self.calculator = Calculator()

    def test_add(self):
        a = 10
        b = 20
        expected = 30
        result = self.calculator.add(a, b)
        self.assertEqual(expected, result)
```

PyNose

**Smells Detected by PyNose**

**test_Calculator.py**

**TestCalculator**

- ConstructorInitialization

# Default Test

A test suite is called MyTestCase which is a default name provided by IDE.

Calculator.py | test_Calculator.py | case.py | Pipfile

```python
from unittest import TestCase

from Calculator import Calculator


class MyTestCase(TestCase):

    def test_add(self):
        a = 10
        b = 20
        expected = 30
        result = Calculator().add(a, b)
        self.assertEqual(expected, result)
```

PyNose

**Smells Detected by PyNose**

**test_Calculator.py**

**MyTestCase**

- DefaultTest

# Duplicate Assert

A test case that contains more than one assertion statement with the same parameters.

Calculator.py | test_Calculator.py | case.py | Pipfile

```python
from unittest import TestCase

from Calculator import Calculator


class TestCalculator(TestCase):

    def setUp(self):
        self.calculator = Calculator()

    def test_add(self):
        a = 10
        b = 20
        expected = 30
        result = self.calculator.add(a, b)
        self.assertEqual(expected, result)
        self.assertEqual(expected, result)
```

PyNose

**Smells Detected by PyNose**

**test_Calculator.py**

**TestCalculator**

- AssertionRoulette
- DuplicateAssertion

# Empty Test

A test case where everything is commented out.

test_Calculator.py | time.py | Calculator.py | builtins.py | case.py | Pipfile

```python
from unittest import TestCase

from Calculator import Calculator


class TestCalculator(TestCase):

    def test_add(self):
        calculator = Calculator()
        # result = calculator.add(10, 20)
        # self.assertEqual(30, result)
```

# Exception Handling

A test case that contains either the try/except statement or the raise statement.

test_Calculator.py | Calculator.py | builtins.py | case.py | Pipfile

```python
from unittest import TestCase

from Calculator import Calculator


class TestCalculator(TestCase):

    def setUp(self):
        self.calculator = Calculator()

    def test_divide(self):
        error_occurred = 0
        a = 10
        b = 0
        try:
            self.calculator.divide(a, b)
        except ZeroDivisionError:
            error_occurred = 1
        finally:
            expected = 1
            self.assertEqual(expected, error_occurred)
```

PyNose

**Smells Detected by PyNose**

test_Calculator.py

TestCalculator

- ExceptionHandling

# General Fixture

Not all fields instantiated within the setUp() method of a test suite are utilized by all test cases in this test suite.

Calculator.py | test_Calculator.py | case.py | Pipfile

```python
from unittest import TestCase

from Calculator import Calculator


class TestCalculator(TestCase):

    def setUp(self):
        self.calculator = Calculator()

    def test_add(self):
        expected = 30
        a = 10
        b = 20
        new_calculator = Calculator()
        result = new_calculator.add(a, b)
        self.assertEqual(expected, result)

```

PyNose

**Smells Detected by PyNose**

**test_Calculator.py**

**TestCalculator**

- GeneralFixture
- TestMaverick

# Ignored Test

A test case that contains the @unittest.skip decorator.

Calculator.py | test_Calculator.py | case.py | Pipfile

```python
from unittest import TestCase

from Calculator import Calculator


class TestCalculator(TestCase):

    def setUp(self):
        self.calculator = Calculator()

    @unittest.skip("Ignored add test")
    def test_add(self):
        a = 10
        b = 20
        expected = 30
        result = self.calculator.add(a, b)
        self.assertEqual(expected, result)
```

PyNose

**Smells Detected by PyNose**

**test_Calculator.py**

**TestCalculator**

- IgnoredTest

# Lack of Cohesion of Test Cases

Test cases that are grouped together in one test suite but are not cohesive.

test_Calculator.py | time.py | Calculator.py | builtins.py | case.py | Pipfile

```python
from unittest import TestCase

from Calculator import Calculator


class TestCalculator(TestCase):

    def test_divide(self):
        Calculator.add(10, 20)

    def test_add(self):
        self.assertTrue(bool(1))
```

PyNose

**Smells Detected by PyNose**

**test_Calculator.py**

**TestCalculator**

- UnknownTest
- LackCohesion

# Magic Number Test

A test case that contains an assertion statement that contains a numeric literal as an argument.

Calculator.py | test_Calculator.py | case.py | Pipfile

```python
from unittest import TestCase

from Calculator import Calculator


class TestCalculator(TestCase):

    def setUp(self):
        self.calculator = Calculator()

    def test_add(self):
        a = 10
        b = 20
        result = self.calculator.add(a, b)
        self.assertEqual(30, result)
```

PyNose

**Smells Detected by PyNose**

**test_Calculator.py**

**TestCalculator**

- MagicNumberTest

# Obscure In-Line Setup

A test case that contains ten or more local variables declarations.

test_Calculator.py | time.py | Calculator.py | builtins.py | case.py | Pipfile

```python
from unittest import TestCase

from Calculator import Calculator


class TestCalculator(TestCase):

    def setUp(self):
        self.calculator = Calculator()

    def test_divide(self):
        a1 = 10
        b1 = 0
        with self.assertRaises(Exception):
            self.calculator.divide(a1, b1)
        a2 = 20
        b2 = 10
        expected2 = 2
        self.assertEqual(expected2, self.calculator.divide(a2, b2))
        a3 = 10
        b3 = 3
        expected3 = 3
        self.assertEqual(expected3, self.calculator.divide(a3, b3))
        a4 = 121
        b4 = 11
        expected4 = 11
        self.assertEqual(expected4, self.calculator.divide(a4, b4))
```

PyNose

**Smells Detected by PyNose**

**test_Calculator.py**

**TestCalculator**

- AssertionRoulette
- ObscureInLineSetup

# Redundant Assertion

A test case that contains an assertion statement in which the expected and actual parameters of equality are the same.

Calculator.py | test_Calculator.py | case.py | Pipfile

```python
from unittest import TestCase

from Calculator import Calculator


class TestCalculator(TestCase):

    def setUp(self):
        self.calculator = Calculator()

    def test_add(self):
        a = 10
        b = 20
        expected = 30
        self.calculator.add(a, b)
        self.assertEqual(expected, expected)
```

PyNose

**Smells Detected by PyNose**

test_Calculator.py

TestCalculator

- RedundantAssertion

# Redundant Print

A test case that invokes the print() function.

Calculator.py | test_Calculator.py | case.py | Pipfile

```python
from unittest import TestCase

from Calculator import Calculator


class TestCalculator(TestCase):

    def setUp(self):
        self.calculator = Calculator()

    def test_add(self):
        expected = 30
        a = 10
        b = 20
        result = self.calculator.add(a, b)
        print("expect 30 for add(10, 20)")
        self.assertEqual(expected, result)
```

PyNose

**Smells Detected by PyNose**

**test_Calculator.py**

**TestCalculator**

- RedundantPrint

# Sleepy Test

A test case that invokes the time.sleep() function with no comment.

test_Calculator.py | time.py | Calculator.py | builtins.py | case.py | Pipfile

```python
from unittest import TestCase

from Calculator import Calculator


class TestCalculator(TestCase):

    def setUp(self):
        self.calculator = Calculator()

    def test_divide(self):
        a = 10
        b = 0
        time.sleep(1)
        with self.assertRaises(Exception):
            self.calculator.divide(a, b)
```

PyNose

**Smells Detected by PyNose**

**test_Calculator.py**

**TestCalculator**

- SleepyTest

# Test Maverick

A test suite contains at least one test case that does not use a single field from the SetUp() method.

Calculator.py | test_Calculator.py | case.py | Pipfile

```python
from unittest import TestCase

from Calculator import Calculator


class TestCalculator(TestCase):

    def setUp(self):
        self.calculator = Calculator()

    def test_add(self):
        expected = 30
        a = 10
        b = 20
        new_calculator = Calculator()
        result = new_calculator.add(a, b)
        self.assertEqual(expected, result)

```

PyNose

**Smells Detected by PyNose**

**test_Calculator.py**

**TestCalculator**

- GeneralFixture
- TestMaverick

# Unknown Test

A test case does not contain a single assertion statement.

Calculator.py | test_Calculator.py | Pipfile | case.py

```python
from unittest import TestCase


class TestCalculator(TestCase):
    def test_add(self):
        self.fail()

    def test_subtract(self):
        self.fail()

    def test_memoize(self):
        self.fail()

    def test_clear(self):
        self.fail()
```

PyNose

**Smells Detected by PyNose**

**test_Calculator.py**

**TestCalculator**

- UnknownTest

# Identifying Python specific test smells

- Selected a **primary dataset** of mature open source projects with atleast **1000 commits**, 10 contributors.

- Find all test files having **"test"** in its name. which is the **unittest** naming convention.

- Find patterns in the histories of the collected projects where changes made to test files that might be considered as **fixing the test or fixing the smell**.

- A Tool developed to mine change patterns in commit history is named **PythonChangeMiner**.

# PythonChangeMiner

- It builds a **program dependence graph** with representing the code by showing its **data dependencies** and **control dependencies**.

- It builds a **change graph** for the fragment of code changes using the versions of code **before** and **after** the target change.

- It **mines** such change graphs from git repositories by traversing their **commit history**, and **discover patterns** in these change graphs.

- The pattern is defined by two thresholds: **minimum number of graph nodes** in the pattern, and **minimum number of repetitions** of the pattern in the corpus.

# Suboptimal Assert

- After identifying common patterns more than 3 different projects, majority of patterns related to assertion functionality changes which are related to test smells.

1. Assertion changes that **alter the logic**. Ex. Changing from **assertEqual to assertRegex**.
2. Assertion changes that **do not alter the logic** and use more **appropriate** functions. Ex. **assertTrue(X in Y) to assertIn(X, Y)** or **assertEqual(X, False) is changed to assertFalse(X).**
3. Assertion changes that **do not alter the logic** and **use less appropriate functions**. Ex. Moving from a more specific **assertIsNotNone(X)** to a more general **assertNotEqual(X, None)** which leads to test smell.

# Suboptimal Assert

A test case that contains at least one of the suboptimal asserts.

test_Calculator.py | time.py | Calculator.py | builtins.py | case.py | Pipfile

```python
from unittest import TestCase

from Calculator import Calculator


class TestCalculator(TestCase):

    def setUp(self):
        self.calculator = Calculator()

    def test_previous_results(self):
        self.calculator.add(10, 20)
        self.calculator.divide(8, 3)
        calculator_results = self.calculator.results()
        self.assertTrue(30 in calculator_results)
```

PyNose

**Smells Detected by PyNose**

**test_Calculator.py**

**TestCalculator**

- SuboptimalAssert

# PYNOSE

- **PYNOSE** is implemented as a **plugin** for **PyCharm** IDE.

- This tool identifies 18 test smells in actual Python code. It parses .py files to PSIFile objects and detects smells.

- Supports **GUI mode** in active code development or **CLI mode** for offline analysis.

- It can report the results as **JSON files** for further evaluation.

# Evaluation of PyNose results

- Manually selected 8 projects that **did not make** it to **primary dataset** and marked as **validation set**.

- Then the validation are run against **PYNOSE** and results are compared against the manual set.

- The results like **precision**, **recall** and **F1 values** are compared with manual and looks very close. (94.0%, 95.8%, 94.9%)

- Also the overall results are compared with **TSDetect** (similar tool for Java). (96.0%, 97.1%, 96.5%)

# Empirical study on test smell prevalence

- Apart from **primary dataset**, **secondary dataset** of projects (less commits) was selected for identifying test smells.

- The **purpose** of the **secondary dataset** is to make sure that the reported results are **unbiased**.

- Test smells **distribution** are similar when compared between **primary dataset** and s**econdary dataset.**

- For better analysis, **co-occurence of test smells** are identified across test suites. The test smell **distribution percentage** demonstrates that test smells have **relationship** with one another.

# Future work

- Supporting more **PYTHON** test smells.

- Conducting a more thorough comparison of **PYNOSE** to other tools. For example, to **TSDETECT** that works with Java/Scala.

- It would be of **great interest** to see how **test smells correlate** with **test coverage**.

# Thanks!