# 4. Dynamic Analysis: Testing

(Last Update: 2024-02-28)

Dynamic analysis is "the analysis of the properties of a running software system" [Ball1999]. It is complementary to static analysis techniques. Some properties that cannot be studied through static analysis can be examined with dynamic analysis and vice versa. The applications of dynamic analysis techniques are very broad: program comprehension, system verification, resource profiling, test analysis, etc. In this session, we focus on one very important aspect of dynamic analysis: Testing.

As the chapter so eloquently states, "Tests: Your Life Insurance!" (OORP, p.149). Tests are essential for Reengineering activities. They can help you: (1) to reveal unwanted side effects of refactoring ("Write Tests to Enable Evolution", OORP, p.153); and (2) to understand the inner workings of a system ("Write tests to Understand", OORP, p.179).

The presence of automated tests does, however, not offer any guarantee about its quality. Do the tests cover the whole system or are some parts left untested? Which parts are covered to which extend? Hence, measuring test coverage is a useful, even necessary, way to assess the quality and usefulness of a test suite in the context of reengineering.

**Sample Projects and Tools (**Check slides here.**)**

Projects

- pacman-python (https://github.com/kilincceker/pacman-python-ua-sre)
- django CMS (https://github.com/django-cms/django-cms)

IDE

- PyCharm CE (https://www.jetbrains.com/pycharm) – You can use community edition or other IDEs.

Tools

- Coverage.py is a code coverage tool supported by PyCharm (you can use coverage tool at your discretion, but it may require some adaptations for the project we are using during the lab sessions).
- MutMut is a tool to perform Mutation testing on Python. Mutation testing is a relatively new topic; therefore, it is difficult to find working tools for it.

Book

- Object-Oriented Reengineering Patterns (OORP - http://scg.unibe.ch/download/oorp)

**Auxiliary Tools**

Auxiliary tools are not required for the lab session itself, but they may be useful to get additional information (or alternatives) on a project. Use them at your own discretion.

- SonarQube is a tool/platform that performs static analysis on source codes. It shows test coverage data from the output of dynamic analysis tools.
- AmPyfier is a tool for test-amplification in Python. Test-amplification generates new test cases using the original tests as seeds.
- Pynguin is a test generation tool for Python. It automatically generates unit tests.
- ChatGPT is a large language model. It can be used for test generation and test amplification.

**Setup / Preparation**

Be sure to follow the setup and the tasks from the last session (Refactoring Assistants). Especially the task of executing SonarQube. As usual, you will also need PyCharm and projects. Moreover, if you have not already, download the book for this course, *"Object-Oriented Reengineering Patterns"*. Get/Install the tools & plugins detailed in the Materials above (not the auxiliary tools, they are optional and will not be used in this lab session).

**Task 1 – Coverage.py Test Coverage**
We will begin by using the Coverage.py test coverage tool. The testing and coverage tools should be enabled by default.

First, make sure that you can test your Pacman-python, by using the following command line in the PyCharm terminal:

To run the tests (+ coverage) for unit test:

```
python -m coverage run refactored/pacman_unittests.py
python -m coverage html
python -m coverage erase
```

And for integration test:

```
python -m coverage run refactored/program_integration_tests.py
python -m coverage html
python -m coverage erase
```

This should provide you with a html coverage report with statement coverage under htmlcov directory.

Rename the generated folder for the unit and integration test you run for statement coverage. However, there are other coverage metrics available for you to analyses the adequacy of your test.

Alternatively, you can use branch coverage and include this in your html report using Coverage.py that supports statement coverage by default. To expand your report with branch coverage, check the documentation below.

To run the tests (+ coverage) for unit test:

```
python -m coverage run --branch refactored/pacman_unittests.py
python -m coverage html
python -m coverage erase
```

And for integration test:

```
python -m coverage run --branch refactored/program_integration_tests.py
python -m coverage html
python -m coverage erase
```

If everything is executed without errors, you should see a new file showing the code coverage in html file under htmlcov directory. Rename the generated folder for the unit and integration test you run for branch coverage. Please try to remember this coverage (or take a screenshot to not depend on your memory).

Questions

- Is the coverage good enough?
- If you make any changes in Pacman-python sources, can you rely on the current tests to catch faults?
- Are the coverage results from statement coverage similar to the ones you got from branch coverage in the last task? Why so or why not?
- Can you apply these to django CMS project? What are your observations?

**Task 2 -- Increasing Coverage on Pacman-python**
For the second task, we will increase the statement and branch coverage on Pacman-python. Doing that is very simple, we just need to write more tests. In this task, we are going to write one new test case.

Let's create a simple unit test on a method. We will test the " move_ghosts" function in " pacman.py" file. You should look at the " pacman_unittests.py" file as a template for your test case.

After adding the new test, run "pacman_unittests.py" again and run it with coverage. If your test has no errors, you should see the coverage report showing the code coverage. Leave this report with the coverage information on as you may need it to answer the questions from the next task (or take a screenshot of it).

Questions:

- How did the new test affect the coverage?
- Do you think 100% coverage is feasible?
- What would you propose as a good level of code coverage?

**Task 3 -- Mutation Testing on Pacman-python**

Mutation testing is a method of determining the quality of a test suite in detail. Mutation testing simulates faults and checks whether the test suite is good enough to catch those simulated faults. It is performed by injecting faults into the software and counting the number of these faults that make at least one test fail. The process for mutation testing requires the following steps. First, faulty versions of the software are created by introducing a single fault into the system (Mutation). This is done by applying a known transformation (Mutation Operator) to a certain part of the code. After generating the faulty versions of the software (Mutants), the test suite is executed on each one of these mutants. If there is an error or failure during the execution of the test suite, the mutant is marked as killed (Killed Mutant). On the other hand, if all tests pass, it means that the test suite could not catch the fault and the mutant has survived (Survived Mutant). Mutation testing demands a green test suite — a test suite in which all the tests pass — to run correctly. The final result is calculated by dividing the number of killed mutants by the number of all mutants. A test suite is said to achieve full test adequacy whenever it can kill all of the mutants. Such test suites are called mutation-adequate test suites.

For the lab, we recommend using the tool MutMut to perform mutation testing on Python. This tool is open-source and already utilized for several industrial projects. MutMut is a mutation testing tool written in python that can analyze a wide range of Python applications.

To install MutMut follow the instructions on its GitHub page. On Mac/Linux already with python you just need to use the following command to install it:

```
pip install mutmut
```

In this task, you will try mutation testing on Pacman-python. Install MutMut and run it on Pacman-python source code. Mutation testing is a slow procedure, the process may take some time. Therefore, don't "kill" the process if you think it is taking too long (let the tool do its job). Please check the instruction below very carefully and follow it based on your project.

https://github.com/boxed/mutmut/blob/master/README.rst

After it is finished, you can find the report in a structure like "/html/index.html". This is like a coverage report but instead of showing coverage percentage, it shows Mutation Coverage (i.e., the percentage of mutants killed). A high mutation score indicates that your test suite is a good quality one. Take the statement and branch coverage reports or SonarQube results on coverage and compare it with the mutation coverage.

Questions

- Find classes that have low mutation coverage and high statement coverage. What does this indicate?
- In these classes, find a survived mutant that is within a covered statement (you can use the line numbers and the before-after comments inside each mutant to find them in the code). Why does this happen?

- Find classes that have high mutation coverage and low statement coverage. What does this indicate?
- In these classes, look for a killed mutant that is not within a covered statement. Why does this happen?
- Given the previous observations, do you think statement coverage is a reliable metric for the quality of the test suite? Why (not)?
- Can the accuracy of mutation testing be improved? If yes, how do you think it is possible?
- How can mutation testing help write new tests?

**SonarQube Coverage Information on Pacman-python**

SonarQube is a static analysis tool. But it can show the results from test coverage in its interface. First, let's run SonarQube as is defined in lab exercises 3 (Refactoring Assistants). Make sure your SonarQube service is running. Please see the documentation below for a coverage report from SonarQube.
https://docs.sonarsource.com/sonarqube/9.8/analyzing-source-code/test-coverage/python-test-coverage/

Questions:

- What do you think of the overview coverage visualization provided by SonarQube?
- Which did you find better to visualize the source code with statement coverage, branch coverage or SonarQube?

**Test Amplification and Test Generation: Increasing (Even More) the Coverage on Pacman-python**

As an optional task, try to increase the coverage even further on Pacman-python. Look for places with lower coverage and create new tests for those using AmPyfier, Pynguin, and ChatGPT. This could help you understand better the application as explained in the pattern "Write tests to Understand" (OORP, p.179).

Questions:

- Did your knowledge of Pacman-python inner structures improve after creating new tests for it?
- Do you feel more confident in changing the artifacts where you increased their coverage?