TestInProd - Project Report

Topics in Software Engineering

Nigel Schuster, Will Pascucci

Introduction

Testing is an essential part of software engineering, and robust test practices help projects find critical bugs and prevent failures. Testing surfaces a wide range of issues and often prevents regressions. Test practices have evolved as much as software engineering has evolved over the past few decades, beginning with manual testing, evolving into automated test suites, and growing now into a critical aspect of DevOps processes. Nowadays, we have a wide range of tools available to enhance our testsuite, including Selenium, Snapshot Testing, Mutation Testing and more. Yet, the fundamental practice of writing a testcase has barely evolved, since the beginning of testing.

Developer's often perceive testing as another task on their respective to-do lists and can assign it a lower priority. In the absence of strategies like Test Driven Development, writing testcases can be done after much of the development work as been done. This project aims to fundamentally alter the approach of writing testcases. The first change is to generate testcases from on previous manual executions. This is based on the following observation: Before writing a testcase a developer usually tests the code manually (excluding functional testing). Given a successful execution, the captured input allows us to immediately generate a regression testcase. A developer can choose to alter the code afterwards, but he has a working testcase available already. The second change we propose regards improving the testsuite. Instead of the developer using code coverage or mutation testing separately to improve his testsuite, we want to utilize the tools directly to proactively suggest improvements. Finally, we do not believe that testing is another step in developing a software component, but it is intertwined with the development of the actual code. Therefore, "TestInProd" aims to be a testing framework that continuously provides developers with an intuitive, low-effort way to continuously build up a comprehensive set of testcases throughout their development.

Project Goal

The goal of team TestInProd is to augment an existing test framework to add features that will improve the developer experience of testing. For unit and integration testing one still selects concrete scenarios that should be tested, mocks peripheral components, and writes the code to test the scenario. This valuable manual effort is perceived as onerous, and is often left until later in the development process. Techniques such as Fuzz or Metamorphic Testing provide a step ahead, but often only serve in a supplemental nature and may not be frequently used. TestInProd's goal is to build a tool that leverages a developer's existing interaction with their software while supporting techniques such as Fuzz and Metamorphic Testing to provide developers with a simple way to generate more comprehensive testcases with minimal effort to encourage more thorough and frequent unittesting.

Research Questions

The project aims to answer two research questions:

1. Can the manual executions of programs be captured and turned into valuable and extensive unit testcases through a simple interaction such as a method annotation?
2. Would the use of a tool that captures manual executions to generate testcases improve developer's perception and use of testing?

System Description

TestInProd is a testing framework built on top of PyTest that aims to improve the developer interaction when writing testcases. The system provides python decorators as part of the framework which allows developers to annotate classes for testing. Python allows for deep code introspection, making the capturing of all the behaviors and external calls easier than it might be in various other languages. Annotated methods have their interaction captured when they are used and capture the inputs, outputs, and external interactions so a testcase can be written. This leverages a developer's manual interaction with their program and the examples they wrote to auto-generate testcases to reduce the burden of later testing the program.

The system has a number of modes to modify the overall user interaction. The system provides a "thorough" mode that allows a developer to enable support for less traditional testing techniques such as fuzzing and metamorphic testing. This allows a level of granular control to determine the appropriate amount of testing for certain methods. The system also has a "trusted" mode in which all of the executions that are occuring are considered valid testcases. Otherwise, a command-line interaction occurs to prompt the user to determine if an interaction is valid or not to prevent bad executions from becoming testcases.

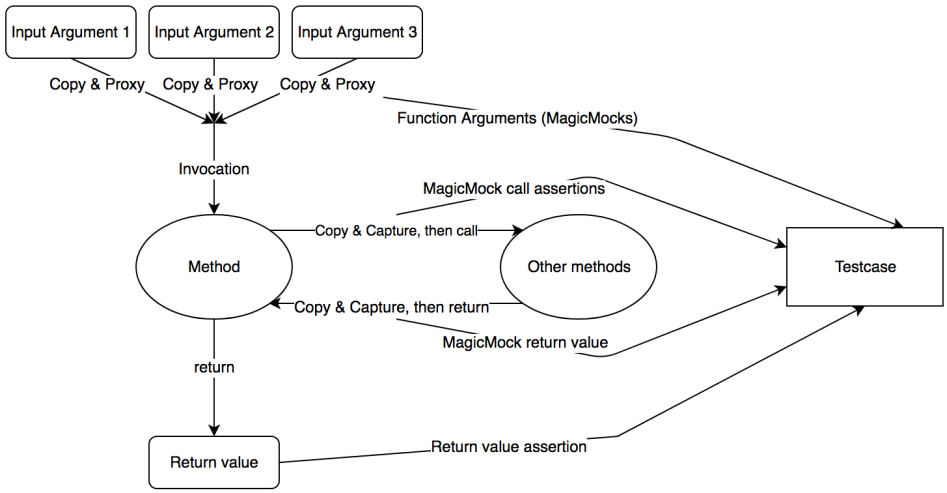

Figure 1. Testcase capture architecture

Project Features

Value Proxies
A major hurdle in the design is to track necessary in- and outputs of a function. Especially if objects are passed to a method we still have to provide guarantees of correct execution. To address this problem we use Proxy Objects. A proxy simply wraps an existing object and forwards all interactions with it to the object. This structure allows us to easily capture call data while introducing minimal additional complexity. Python allows us to achieve this in very few lines of code. A latent challenge to Value Proxies is the representation of its data. We chose to implement our own crude serializer, that turns Python data structures into Python code representing the data. To achieve this for Value Proxies, we made extensive use of MagicMocks. This model has proven successful in that it can reconstruct input and output data with minimal impact.

Thorough Mode
Thorough mode is a separate annotation for the developers classes that will generate a larger number of testcases than just the interactions that are captured. Developer interactions are a highly valuable means of determining the correct functioning of newly written code, but often times the interactions may be limited. Anecdotally, developers may verify the correct functioning of a newly written class with normal-case execution without expressly testing all erroneous inputs or edge cases. Thorough mode attempts to augment those normal-case interactions by working with the values provided in such cases. Those values will be examined for type to provide inputs of every other type in a form of fuzz testing or will be minorly adjusted in the case of metamorphic testing.

This feature may introduce a larger computational overhead because it can introduce a variety of manipulations on top of standard testing. For example, a function that expects integers might only need a few valid inputs for testing if it is only accessible by other system code and will likely not receive non-integer inputs. In this case, thorough mode might not be needed and the function's ability to handle strings or booleans does not need to be verified. However, for many functions, it may prove beneficial to see how it handles inputs of every type to verify that no system-breaking error occurs. This testing may introduce a larger computational overhead, which is why it was introduced as an optional mode and not a default behavior.

User Interaction
With the large focus of the project being on reducing the developer difficulty and increasing the developer enjoyment of testing, the system aims to interact with the developer as little as possible. This prompted the creation of a "trusted" mode. If a developer enables a trusted mode, then the developer would not need to interact with the system until they aim to execute those test cases. The system would just monitor the developer-written function calls and capture the behavior to write a test case. This is great for ease of use, but allows for the possibility of capturing incorrect executions. Capturing bad executions and turning them into a testcase would damage the use of the framework. To handle that, a developer can turn "trusted" mode off, which will then interact with the developer via the command line so the developer can

indicate if an execution is correct or not. The main interaction a developer will have with the system will be annotating the classes that he or she wishes to have captured by the system and running the generated test files.

Fuzz Testing
One of the additional testing practices that is supported by the system is the use of fuzz testing to test functions' ability to handle input types that were not covered in the main interactions. This feature was implemented by examining the type of a value for a given execution and generating empty values of the other types. These are then run without an assert check to verify that the function will not crash when it is executed.

Metamorphic Testing
Manipulating given input and output values to generate a new, correct was not something that could be reliably put into the system without user input. The system approaches this issue in a twofold manner: first to apply a minor, non-type changing modification to a value to ensure the method does not crash. Examples of metamorphic changes that represent intuitive errors for a given type include a negative value from an integer (or any numerical) input, a case change (upper or lower) for a string input, additional whitespace added to the end of a string input, a reversed list to a list input, and the opposite value for a boolean input. These metamorphic changes were included in the framework.

Unique Aspects

The system is unique in a number of ways especially regarding the ability to capture and leverage manual executions to build up a base of testcases. Other frameworks may try to simplify writing of testcases, which we believe developers still find to burdensome; while others try to automate testcases, which we do not believe is generally accurate enough. The manual execution capturing with the ability to make fuzzing changes or metamorphic alterations on top of those captured inputs is what makes our testing framework unique.

Our approach will also function on the scale of working with teams by helping each developer without the test cases interfering with others. Generated test cases will not override one another and will be added to increase test coverage. The framework will not react differently to multiple developers working on one class than it would to one developer working on that class multiple times.

Target Community

The target community for this system is virtually all software engineers potentially with the exception of specialized test engineers who would not want the process on which they focus to be automated further. Because writing testcases can seem tedious to developers, it is a step in the development cycle that is often delayed until the end or ignored completely. The system will incrementally gather testcases that a developer may choose to use or not to use, but it will automatically do the work for them just by harnessing manual interactions.

Value Added

The value added by the system is quite immediate and apparent because the developer has existing manual executions that he or she is already doing captured and turned into testcases. This automatically creates valid testcases without any additional burden on the developer except for annotating the method with their preferred annotation. With additional capabilities like thorough mode, a single manual executions can turn into several testcases without any effort. By using valid executions, fuzzing inputs to different types, and changing input to check other logical, same-type values, developers can increase their test coverage and improve the testing of their software.

Evaluation Strategy and Data Gathering

The main evaluation strategy had two aspects: assessing the results of the framework's ability to generate testcases and asking participants to interact with the framework on dummy projects. Participants were then asked to complete a survey.

Testcase Generation
A primary endpoint was the generation of accurate and useful testcases. This was a more difficult metric to gauge because of the sheer number and variety of functions to which the framework could be applied. The project members applied the framework to a baseline dummy project and an existing project from a COMS 4156 class completed the previous semester. From these two programs, a variety of testcases were generated and evaluated.

Participant Recruitment
Participant recruitment was done mostly by asking several students in the CS program to test the system. Students chosen were often former and current classmates of the project team. Most students chosen have had professional experience through their internships and have seen software development and testing at both large and small companies in industry. Because of the students' experience in software development, the chosen participants were believed to be a good gauge of a larger development community's perspective, especially the community of newer engineers.

Survey Design
The survey was designed to assess the participants perception of unit testing, their writing of testcases, and their experience with our tool. The questions were:

1. How would you rate your overall experience with unit testing? (Required)
   Answered with a 1 (Bad) to 5 (Great) scale.
2. When in your development do you write your units tests? (Required)
   Answered with a "Check all that apply" that includes options for before, during, and after development.

3. How frequently do you manually execute your code? (Required)
   Answered with a multiple choice of rarely, sometimes, frequently.
4. How do you rate the experience with our tool? (Required)
   Answered with a 1 (Bad) to 5 (Great) scale.
5. Were the generated testcases accurate and natural? (Required)
   Answered with a multiple choice of yes, no, only some, or most but not all.
6. What did you like most about our tool? (Optional)
   Answered with a long paragraph.
7. Would you be more likely to write unit tests sooner and more frequently with our tool? (Required)
   Answered with a multiple choice of yes, no, or maybe.

Issues During Recruitment and Participation

The main issue with the survey aspect was participant recruitment. This was not because many were not eager to see the framework; it was because many potential participants were highly busy during the final weeks of the semester and could not take the time to work with the framework. Once participation began, the process was smooth.

Evaluation Metrics

The project was evaluated based on the ability to generate accurate and useful testcases, the amount of effort required to use the tool, and the change in the participants' perception of writing testcases. The ability to generate test coverage was evaluated on the coverage of the executed statements. This metric is a good metric, but has limitations due to the wide variety of functions to which it could be applied.

More objective evaluation came from the results of the survey where the developers' explicit experiences were recorded and used for results.

Results

Test Coverage Generation

We started by applying our tool to a dummy project with just one class and two methods. That allowed us to learn about initial problems of the tool as well is quickly iterate. Once we achieved 100% test coverage for various scenarios in the dummy class, we considered our tool ready to be applied to a real project.

The testing framework was applied to a previous project from a COMS 4156 class that implemented a web server that acted as a stock trading simulator. When the framework was applied to a subpart of that project, the framework successfully captured the methods that were executed. This demonstrated that the framework could be applied in a non-trivial case of a web server and still provide the desired functionality. The resulting testcases achieved 73% coverage of the classes in the application.

Figure 2 show a code snippet of a method from the COMS 4156 application with Figure 3 showing the generated testcases.

```
def available_cash(self, update=True):
    """
    Returns the available cash for the trading account
    """
    return (self.trading_balance() + sum([
        bnk.current_balance(update)
        for bnk in self.profile.user.userbank.all()
    ]))
```

Figure 2. Method in annotated class

```
def test_available_cash_1(self):
    trading_balance = MagicMock(return_value=0)
    current_balance = MagicMock(return_value=890.0)
    arr_iter = MagicMock(current_balance=current_balance)
    all = MagicMock(return_value=iter([arr_iter]))
    userbank = MagicMock(all=all)
    user = MagicMock(userbank=userbank)
    profile = MagicMock(user=user)
    arg = MagicMock(trading_balance=trading_balance, profile=profile)
    assert 890.0 == TradingAccount.available_cash(arg, False)
    trading_balance.assert_has_calls([call(*[], **{})])
    current_balance.assert_has_calls([call(*[False], **{})])
    all.assert_has_calls([call(*[], **{})])
    assert trading_balance.call_count == 1
    assert current_balance.call_count == 1
    assert all.call_count == 1
```

Figure 3. Resulting testcase

Survey Results

Unfortunately, due to finals, survey turnout was low. However, we were still able to get valuable feedback. We were able to get 3 participants, but those participants used the framework on actual programs. Participants averaged a response of 3 (on a scale of 1-Terrible to 5-Great) about their experience of unit testing in general, indicating a neutral perception of their experience with testing. Figure 4 shows the responses to the question of when in development participants wrote test cases, showing 66% of participants wrote test cases after code has been written.

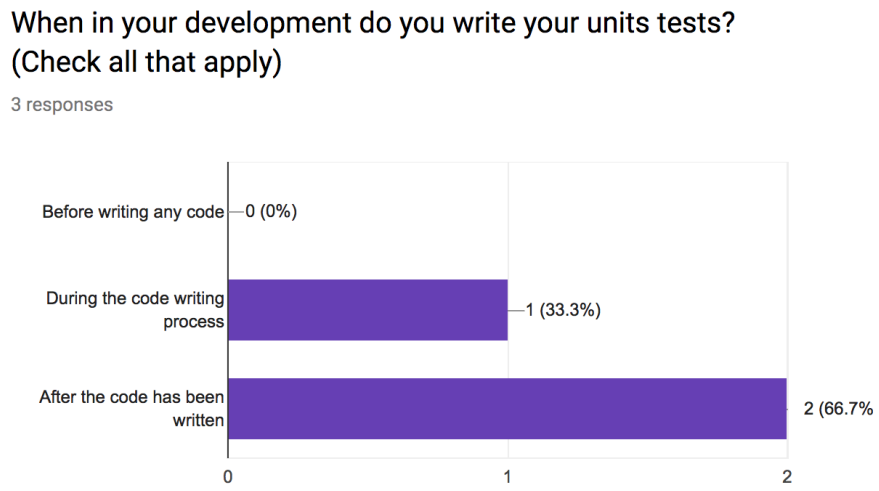

Figure 4. Survey responses to timing of testcase writing

Figure 5 shows 66% of participants indicating that they “frequently” manually executed their code.

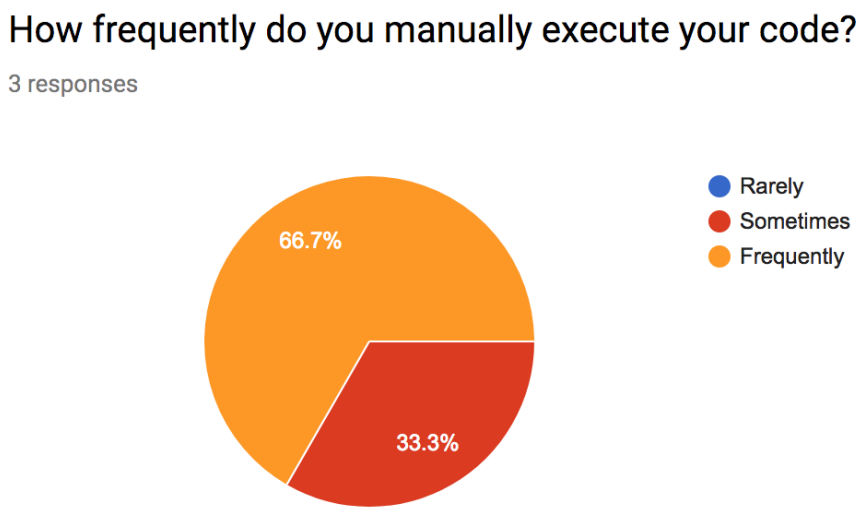

Figure 5. Frequency of manual execution

Responses were split evenly across “Yes”, “Most, but not all”, and “Only Some” regarding the accuracy of the testcases, shown in Figure 5.

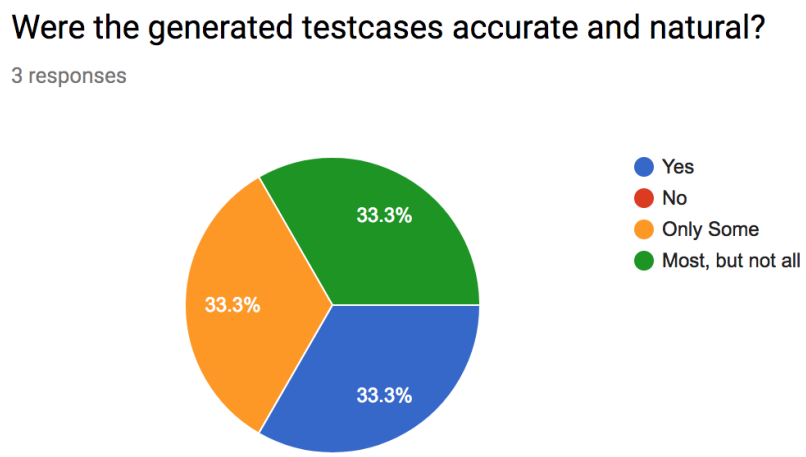

Figure 6. Accuracy of generated testcases

Average response values were 1.33 (on a scale of 1-Not difficult to 5-Very difficult) to the question regarding the ease of use of the framework. Optional text responses to what participants liked the most about the tool included text such as “It was surprisingly easy to use!” and “Its a great idea and can really help save overall development time”. Finally, 100% of respondents indicated that they would be more likely to write unit tests sooner and more frequently with the tool.

Answers to Research Questions

1. *Can the manual executions of programs be captured and turned into valuable and extensive unit testcases through a simple interaction such as a method annotation?*
   Answer: Yes, by implementing a decorator that tracks the function calls of a class and captures the inputs and outputs of that function call, the framework was able to use the manual execution of programs. These inputs and outputs were able to be mapped to test files of unit cases. All that is required of the user of the framework is annotating the desired class with the option of specifying certain parameter keywords.
2. *Would the use of a tool that captures manual executions to generate testcases improve developer’s perception and use of testing?*
   Answer: Yes, by surveying participants and gauging their experience in using our tool, it was determined that the capture of manual testing improved developer’s perception of the testing process. From the participant’s responses, feedback was unanimous in saying that the use of our tool would increase the frequency unit testing.

Discussion

The overall proof of concept of the methodology was successfully achieved. Execution capture served as a useful technique for testcase creation. Moreover, the use of a decorator allows for minimal effort by the developer which continuously builds up testcases as more executions are performed with new methods being added. When applied to real classes, both from the TestInProd team and participants, the framework was able to generalize and still provide valid

and natural testcases. Considering the short time frame for the project, the project serves as a useful base point for the methodology and could be further developed in a more robust capacity that could further build the statement coverage and accuracy of the test cases.

Interestingly, developers rated their experience more highly than was anticipated, with the average respondent values hovering in the middle of the provided scale. Developers responded that they had an easy experience in using the framework which falls in line with the expectations of team TestInProd. Interestingly, some developers noted that initial setup of the framework was more difficult, but all said that once the framework was set up their use was simple. Respondents used the framework on previously used projects, ranging from graphics applications to robotics. The framework was still able to provide useful testcases, but the responses ranged from “Yes”, “Most, but not all” to “Only some”. While TestInProd believes this result is somewhat expected given the timeframe, it does provide room for improvement for future development.

Team Member Contributions

Nigel
Nigel has worked on the core functionality of the framework. Nigel had the base idea of capturing manual developer interactions, and he has built the main functionality that provides the decorator that developers can use on their classes. He has also handled major hurdles object proxying, dependency tracking, special case in introspection handling, and other core pieces.

Will
Will has worked on several of the features that add on to the core functionality to provide specialized test methods and user interaction. Will built the thorough mode, fuzz testing, metamorphic testing, and duplicate testcase catching which build on the main testing strategy. Will also worked on the user interaction aspect in the case of non-trusted mode.

What Was Learned

Nigel
For me this project was mostly about taking a fun idea and actually implementing it. In retrospect I am surprised how well it worked. I learned a lot more about python intrinsics and about prototyping well. With the limited timeframe it was a challenge to build a product that can show off the idea. Further, working in a team was also in some aspects challenging. Luckily our contributions were mostly orthogonal. While I was working on better capturing input and output, Will was able to work with the initial core and iterate on top of that.
Overall, I am very happy with the product and the feedback. I plan to pursue this project over the summer.

Will

For me, I learned a lot about Python's ability for code introspection and the ability to gain access to sufficient information form the call in a usable way. I also learned about the uses of metamorphic and fuzz testing, methods about which I had never previously heard. I enjoyed working in the team environment. I often like to control the full extent of new projects I work on, but it was a lot of fun being able to rely on the core functionality that Nigel provided and getting to focus on different aspects like user interaction and additional testcase generation.

Unexpected Aspects

Challenges
As the framework was generating test cases, various tests were repeated. This was especially true during the use of thorough mode where fuzz changes and metamorphic changes would overlap. Given that one initial goal of the project was to minimize testcases, the issue of fixing duplicate test cases arose as a challenge.

For the metamorphic changes made in thorough mode, an ideal goal was to execute those changes and prompt the developer with the output. If the developer marked the output as correct, that testcase could be added. However, because the class that was being tested was wrapped with the framework's decorator, executing the decorator led to an infinite recursive call of the class repeatedly calling itself.

Static methods / Object creation posed problems, because our mental model treated a function as just having inputs and outputs. It did not take into consideration that some method calls are not done on the input arguments (e.g. we tracked calls on instance methods, but not static method calls).

One of our key value propositions was that me generate unit testcases that are self contained and only target a small chunk of code. Therefore we were required to understand function calls within our system and need to be able to mock those. Traditionally this is achieved by supplying mocks as function arguments. However we needed to know what to mock and what values to return.

Solutions To Challenges
Testcase minimization was achieved by tracking the unique identifying information of a method, its inputs, its keyword arguments, and its class name. This identifying key was stored in a set during the portion of the code that generates the strings for the testcases that are then written to file. When a new function call is being examined, it is checked against the set. This solved the issue of duplicate testcases and helped achieve a certain level of testcase minimization.

The metamorphic execution challenge was not yet solved and is further described below.

To allow for true unit testing we chose to pursue mocking. In order to know what to mock, we wrapped each argument to a function in a custom class. That custom class forwaded any calls

to the underlying object, but recorded call data. Creating a working mock then only required us to rebuild the recorded data.

Planned Features Not Completed

Static Methods / Object creation
Static methods were an unforeseen challenge for our application. Our initial plan was to capture any input and output of a method by wrapping all inputs with a Proxy. However, static method calls and the instantiation of an object cannot be captured this way. Therefore we were not always able to isolate a class and remove side effects. Instead static method calls were executed even during testcases. If an object was created and returned, then our framework threw an exception. In the time frame given, we were not able to implement a sensible solution.

Execution of Metamorphic Changes
The metamorphic changes to function inputs proposed by the framework represent potential new testcases. The difficulty of this feature comes in the developers trust of the output to such a metamorphic change. For example, if a developer's method does not correctly handle a negative integer and that developer's executions only used positive integers, the framework would input testcases with negative values. However, those new test cases cannot be trusted by default without the developer reviewing them first (otherwise it would compromise the correctness of the testcases). The difficulty in implementing this execution arose when trying to execute the annotated function again from within the decorator. This led to a recursive calling of the decorate calling itself for every execution. This difficulty was not easily overcome and resulted in not completing the usage of metamorphic changes as correct testcases (with assertions on the output).

Demo

Summary
The demo of the project occurred in class on Tuesday, April 24th. During the demo, the main idea of the project was restated to the class. The rough approach for capturing executions, mocking objects, generating fuzz cases, creating metamorphic cases, minimizing testcases, and interacting with the user were explained. An existing project from a previous COMS 4156 class was introduced and had its functionality explained. The software was then run and the testcases were generated. Several of the generated unit testcases were then examined to show the approach for mocking and the validity of the generated testcase.

Concerns
Much of the value of this framework comes from a developer's specific feel and perception of using the framework. While the demo successfully showed the class about the primary functionality of the approach, the impact of actually using the framework was more difficult to convey in a presentation-style format.

Limitations

The framework has a number of limitations mentioned in previous sections regarding certain uses. However, as a proof of concept, the project demonstrates the validity of the approach. Moreover, each of the areas of improvement has a clear strategy for mitigating the issue and does not present a significant barrier to future development of the framework.

The survey addresses the important areas which the tool aimed to improve upon. However, given the proximity of the project completion to finals, the survey was limited by the small number of respondents. More participants had been recruited, but many could not find time during finals season.

Conclusion

Testing will always be a crucial aspect of the of the software development process especially in industry environments where errors and bugs can be translated into a direct, tangible cost to a company. Despite this, developers often still view testing and the process of writing and maintaining test suites as an onerous process, potentially resulting in delayed writing of testcases and a reduction in the number and range of the written testcases. The project of team TestInProd was to write a framework that could make the process of writing testcases easier and more approachable for developers by leveraging developer's manual execution of their code and providing simplified support for testing approaches such as metamorphic fuzz testing.

The framework was able to achieve good levels of test coverage on the projects on which the team applied it. On simple, dummy classes, the framework was able to achieve 100% coverage. On a more realistic project, a previously done web service, the framework was able to achieve 73% coverage. The result was pretty strong given the minimal effort (and constant time of annotating a class) required to use the framework. TestInProd felt that this was a very strong level and was an encouraging result.

The survey participants tested the framework on real applications from previous projects and were able to generate useful testcases. Participants reported that they used manual execution in their development, validating the premise of execution capture as a means of building testcases. All participants rated low difficulty in using the tool and rated that the framework would encourage them to write testcases earlier and more frequently.

TestInProd serves as a useful proof of concept for using execution capture as a means of automatically generating testcases to improve the experience of writing testcases.

Deliverables

Access
The project is available on GitHub at the link: https://github.com/Neitsch/pytest. Anyone may clone and employ the framework with their software, given that they are using Python 3. The

developer survey is available at link:
https://docs.google.com/forms/d/1_LWTapxLV1B6nfDAh6VaF_FgEdRNgi-RpGIqEpniX3A/.

Execution Instructions (Also in README)
Prerequisites:

- Python3 must be used for the project you wish to test and for use of the framework.
- Pip (a tool for installing python packages) is strongly encouraged.
- Autopep8 (a tool for formatting the written testcases) must be installed. Can be easily installed with "pip3 install autopep8"

Usage:

- Clone the project into your project directory directory.
- In the file in which you want to use the framework, import "track_class" from "pytest.test_in_prod"
- Annotate the class you wish to track with "@track_class()".
- Now run an execution on the functionality that was annotated. It will result in a unittest file with the name "test_" and the filename.
- For more features, experiment with "@track_class(thorough=True, trusted=False)" to enable thorough mode and disable trusted mode.

Future Work and Development Strategies

To continue development on this framework is very easy. Any developer may fork the project and begin to edit and build on the framework. Pull requests may be created to merge with TestInProd's master branch.

Static Methods / Object creation
As mentioned above, our approach did not allow to capture these actions, since we only look at inputs and outputs of such a function. Instead such method calls need to be captured differently: For static methods we need to wrap each each one with a method that can capture input and output. For object creation we need to modify the __new__ method of each class. This will allow us to catch any new object creation. Thereby we will be able to catch any subsequent method invocations on that object.

Metamorphic Executions
One area that a developer could improve would be the execution of metamorphic test cases. When a metamorphic test case is created, the change to the input should in many cases have a normal, correct result (whereas a created fuzz case just checks to see if different inputs can break the method). Therefore, in non-trusted mode where the developer is interacting with the framework and examining the execution results, the framework can execute a metamorphic change and provide a new testcase that is completely different than the one the developer wrote. The difficulty with this problem though is the recursive nature of the method annotations mentioned above and requires a means of not invoking the decorator used by the framework.

Metamorphic Coverage
The metamorphic changes to method inputs that were included were intuitive ones based on the type of the input value. For example a natural change to an integer would be a negative, to a string would be a upper and lower case string, or to a list would be a reversed list. These changes came from experience with what erroneous inputs to certain variable types might be, but providing the framework with even broader metamorphic changes would be a valuable addition.

Other testing strategies
In addition to the core of the execution capture, the framework aims to provide intuitive access to testing strategies like fuzz and metamorphic testing. A developer could try to broaden the scope of the framework by adding a new type of testing strategy.

Appendix

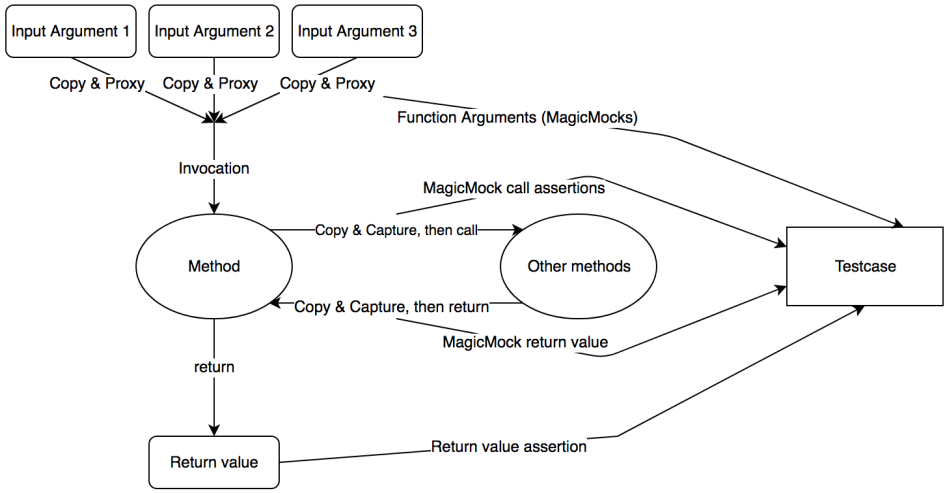

Figure 1. Testcase capture architecture

```
def available_cash(self, update=True):
    """
    Returns the available cash for the trading account
    """
    return (self.trading_balance() + sum([
        bnk.current_balance(update)
        for bnk in self.profile.user.userbank.all()
    ]))
```

Figure 2. Method in annotated class

```
def test_available_cash_1(self):
    trading_balance = MagicMock(return_value=0)
    current_balance = MagicMock(return_value=890.0)
    arr_iter = MagicMock(current_balance=current_balance)
    all = MagicMock(return_value=iter([arr_iter]))
    userbank = MagicMock(all=all)
    user = MagicMock(userbank=userbank)
    profile = MagicMock(user=user)
    arg = MagicMock(trading_balance=trading_balance, profile=profile)
    assert 890.0 == TradingAccount.available_cash(arg, False)
    trading_balance.assert_has_calls([call(*[], **{})])
    current_balance.assert_has_calls([call(*[False], **{})])
    all.assert_has_calls([call(*[], **{})])
    assert trading_balance.call_count == 1
    assert current_balance.call_count == 1
    assert all.call_count == 1
```

Figure 3. Resulting testcase

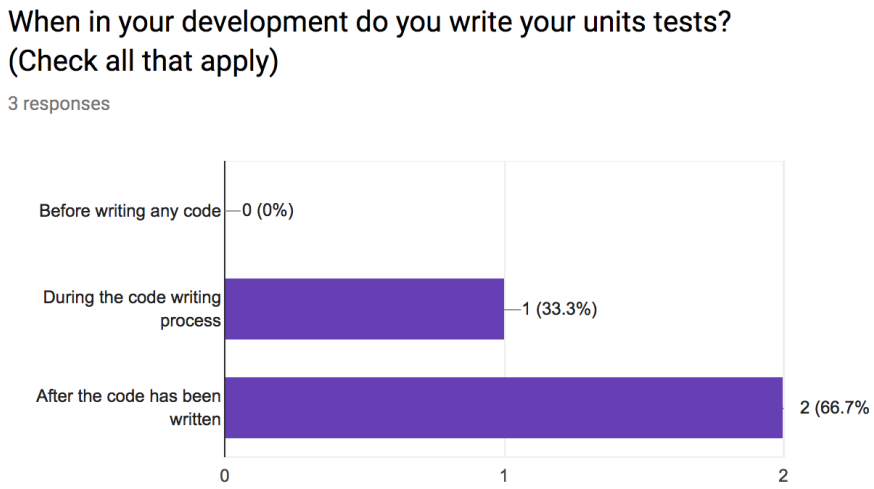

Figure 4. Survey responses to timing of testcase writing

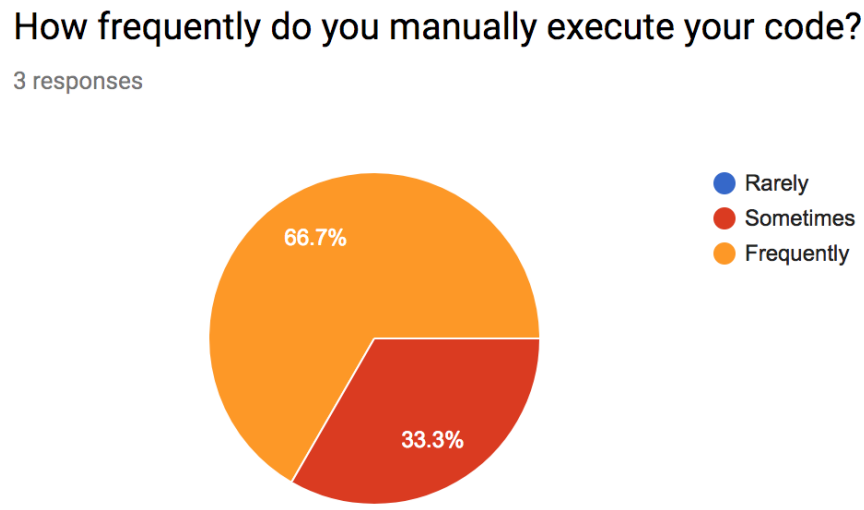

Figure 5. Frequency of manual execution

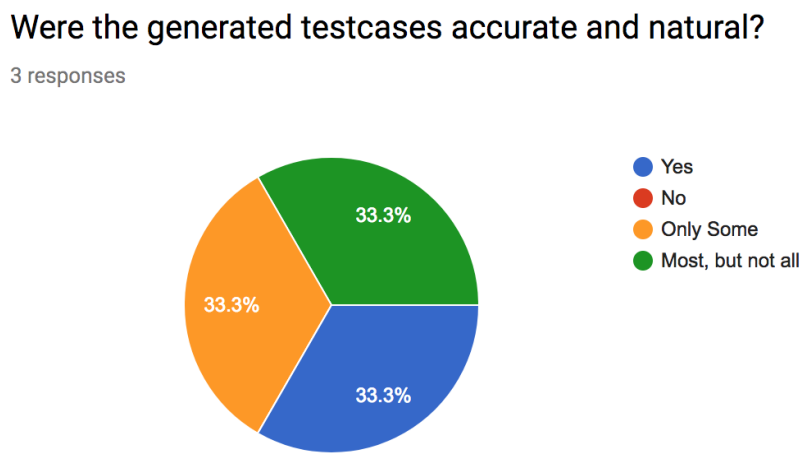

Figure 6. Accuracy of generated testcases

External Tools and Libraries Used

PyTest:
PyTest is the basis of the of TestInProd and is the framework from which TestInProd is forked.
Link: https://docs.pytest.org/en/latest/
Note: While we forked the project, we did not end up needing it explicitly. We planned on integrating deeper with it, but found simpler approaches.

Unittest:
Unittest is a core piece of the testing functionality and serves as the base for the testcases being written as well as provides mocking functionality.
Link: https://docs.python.org/3/library/unittest.html

GitHub:
Git is a highly popular and widely used tool for version control and code sharing. Git (and the site GitHub) was used during the development of the software and is integral to additional developers cloning and using the tool.
Link: https://github.com/

Autopep8:
Autopep8 is used to format the testcases being written to make the written testcases follow the PEP8 style guide as best as possible.
Link: https://pypi.org/project/autopep8/