Proposal for Internal Technical Standards
for 100% Unit Test Code Coverage and Unit Testing of Protected Methods

The purpose of this document is to establish the CFT project specific Technical Standards and Requirements for **Model Validation** in the areas of 100% Unit Test Coverage and Unit Testing of Protected Methods.

The above two areas are a key part of **Model Validation**, which in turn is a key part of the enterprise wide **Model (Risk) Governance framework** of the company.

Model (Risk) Governance and its constituent Model Validation are two areas scrutinized by the regulators, besides the stress test outcomes produced by the models .

Besides that, Model Validation underpins the reliability and quality/precision of Financial Risk Management, which is a mission critical business function for any financial institution, e.g. likes Space Flight Control and Navigation (here we navigate the markets and the economy in way which preserves the integrity and profitability of the company)

Technical Standards for:

a) **Unit Testing of Protected Methods**
b) **What is and how to go about 100% test code coverage**

1. Unit Testing of Protected Methods in a Python Class

(Note: The CFT Model Software Component Framework also contains Protected Python Modules (which makes all python objects defined inside also protected) and the below applies to them too)

(Note: Runtime Access Control for Protected and Private Methods is not enforced in Python, Protected and Private are used to signal the client programmer of CFT Model Component, what is the Public interface of the Component)

Definition of Protected Methods in a Python Class for the purposes of this document:

The CFT project, besides delivering CFT Models, also is delivering a Model Software Component Framework. This is OOP Framework, which facilitates the packaging, distribution and execution of Models in the form of Reusable and Composable Software Components. Ref the next software architecture diagram.

As it can be seen in the software architecture diagram, the CFT Model Software Component Framework and the related MEX Platform, features a two-tier Workflow:

a) Top Tier Workflow - MEX Inter Model Component Workflow
b) Second Tier Workflow – the Internal Workflow the Model encapsulated in a CFT Model Software Component

The organization of the Second Tier Workflow is as follows:

a) The Individual Steps of the Model Workflow are implemented as Protected Methods in the Python Class representing the overall Model – because the individual steps of the Model Workflow should not be called directly by the client programmer / system using a CFT Model Component
b) The orchestration of the Individual Steps is performed from Public Method in the Model Python Class ie this method invokes the protected methods (workflow steps) in the required sequence and with the required business logic associated with that

Considering the above (specifically the two tier workflow and how the second tier workflow is designed and implemented), the convention in OOP, that Protected Methods should not be invoked directly in unit tests should be relaxed when it makes practical sense to do that.

Also it can be seen from the above that testing the Public workflow orchestration method of a CFT Model Component is akin to System Integration Testing (which must be done as well), but for the purposes of an actual Unit Testing, there is a need to incorporate the individual steps of a Model Workflow directly in unit tests.

An alternative way (ie Option 2) of unit testing Protected Methods (instead of invoking them directly in unit tests) – as part of the code of the test suite, create subclasses that simply re-designate protected classes as public for visibility so the testing framework could access otherwise unavailable functions and data members necessary to make assertions on them. This allows testing of the source code and protected members without changing the source code which is a necessity when unit testing.

2. Definition of 100% Unit Test Code Coverage

Note: this area has implications on the overall development effort and hence has to be factored in project plans / ETAs / backlog scheduling

On the surface, code coverage checks **which lines were run at least once** during a test. Tools like gcov perform code coverage analysis by compiling and running the tests, then producing a report.

Note that 100% code coverage does not check which system logic conditions, states etc were tested (within the same piece of code). Most of the coverage tools will report on production lines of code that have been executed. That a line has been called doesn't mean that it has been properly tested (only that it ran). A single statement might encapsulate multiple logical conditions, each of which needs to be tested separately.

The fact that a line was exercised does not mean at all that it was **stressed with all its possible combinations**. The fact that all branches run successfully with the provided data only means that the code **supported that combination**, but it doesn't tell us anything about any other possible combinations of parameters

Some Key Principles (beyond the obvious) when designing unit tests to achieve 100% coverage, while also factoring in the above statement:

- Factor in both, Fair weather scenarios (along with their Edge Cases) and failure scenarios
- Apply Reverse Reasoning from the desired behavior to synthesize an exhausting set of inputs for full coverage and also by injecting faults into the library behaviors
- Exercise all logic conditions and states, system states etc even when the same code branch/block is visited more than once during the unit tests. The previous bullet point about Reverse Reasoning, facilitates the finding of the corresponding sets of test inputs
- Use of Property-based testing - consists of generating data for tests cases to find scenarios that will make the code fail, which weren't covered by our previous unit tests (see more detailed definition next).
- Use of Mutation Testing – failure injection to find suboptimal unit tests

Property-based Testing

Example tools: hypothesis and faker

The following is from the product documentation:

The main library for this is hypothesis which, configured along with our unit tests, will help us find problematic data that will make our code fail. This library can help find counterexamples for our code. We write our production code (and unit tests for it!), and we claim it's correct. Now, with this library, we define a hypothesis that must hold for our code, and if there are some cases where our assertions don't hold, hypothesis will provide a set of data that causes the error. The best thing about unit tests is that they make us think harder about our production code. The best thing about hypothesis is that it makes us think harder about our unit tests.

Another benefit of property based testing is as a way around the monotony of sample data sets - using it as a readily available generator of data entries that could provide realistic values. One such generator is the faker package available on PyPI. faker comes with a built-in pytest plugin, which provides a faker fixture that can be easily used in any of your tests.

Mutation-based Testing

Example tool: mutpy

The following is from the product documentation:

We know that tests are the formal verification method we have to ensure that our code is correct.

And what makes sure that the test is correct? The production code, you might think, and yes, in a way this is correct. We can think of the main code as a counterbalance for our tests. The point in writing unit tests is that we are protecting ourselves against bugs and testing for failure scenarios we don't want to happen in production. It's good that the tests pass, but it would be bad if they pass for the wrong reasons. That is, we can use unit tests as an automatic regression tool—if someone introduces a bug in the code, later on, we expect at least one of our tests to catch it and fail. If this doesn't happen, either there is a test missing, or the ones we had are not doing the right checks. This is the idea behind mutation testing. With a mutation testing tool, the code will be modified to new versions (called mutants) that are variations of the original code, but with some of its logic altered (for example, operators are swapped, conditions are inverted).

A good test suite should catch these mutants and kill them, in which case it means we can rely on the tests. If some mutants survive the experiment, it's usually a bad sign. Of course, this is not entirely precise, so there are intermediate states we might want to ignore.

===----------

$ cd my-project-folder

$ virtualenv --python python3 my-venv

$ source my-venv/bin/activate

PS C:\> cd my-project-folder

PS C:\> virtualenv --python python3 my-venv

PS C:\> .\my-venv\Scripts\activate

===---

When bundling your own packages or projects for other people, you can use:

$ pip freeze > requirements.txt while the virtual environment is active.

==========----------