Administrator's Guide: BMI User Acceptance Testing Framework

Massachussetts Open Cloud

Paul Grosu *(pgrosu@gmail.com)*

First Edition
2017
(Revised on 10/10/2017)

Contents

I

An Approach to Delivering Defect-Free Software Products

CHAPTER 1

Document-Driven Testing for Optimal Deployments

The fundamental principle of science, the definition almost, is this:
the sole test of the validity of any idea is experiment.
— Richard P. Feynman

1.1 How to guarantee a defect-free software product

A software product can be deemed defect-free if all the requirements have been completely covered by tests supporting the specifications, and are continuously validated throughout the development cycle. A development team always strives to provide such guarantees, which can be achieved by being *diligent* in following specific software development practices and ensuring that the requirements are bounded and reflected through *complete coverage*.

One methodology of ensuring that such a continuity is preserved, is via the V-Model of software development[1] illustrated in Figure 1.1.

[1] http://ccdocs.berkeley.edu/content/system-validation-plan

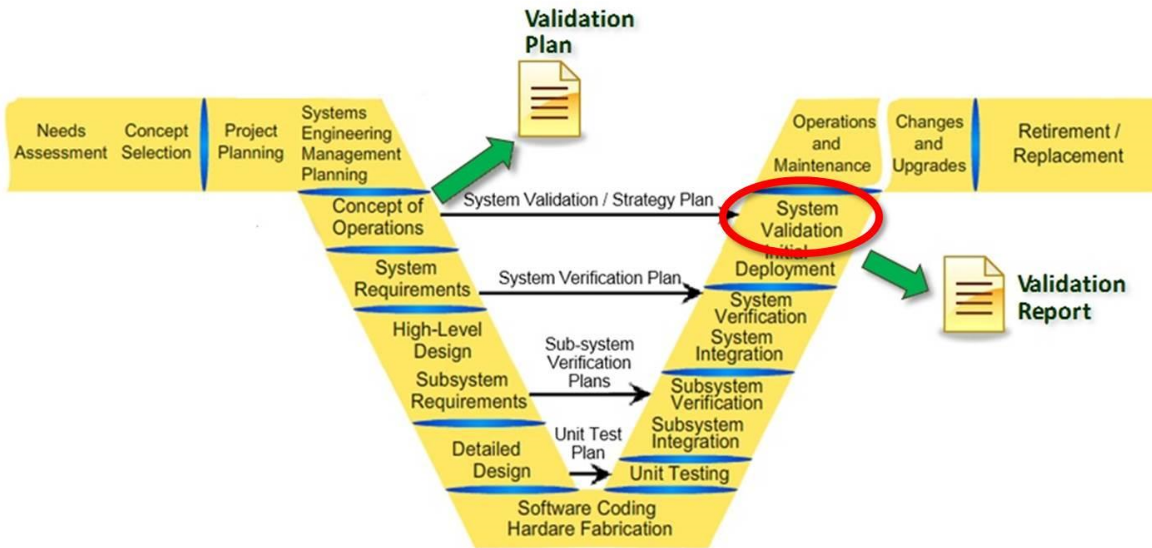


Figure 1.1 – The V-Model of software development.

If you bisect vertically the V-Model, you will notice that each requirement – or implementation – on the left-hand side is *supported* by a verification or validation step on the right-hand side. This ensures that the scope is bounded and provides complete coverage[2]. For every step documentation is critical to both *specify* the requirements, and then to subsequently *validate* against those requirements.

1.2 TESTING UNDER CHANGING ENVIRONMENTS VIA SYSTEM TESTING

There will be times where *Integration Testing* is not enough. This is where *System Testing*[3] comes in. Here we take the software product as a *black box* – as opposed to in Integration Testing – and test it under different environments without touching the code-base. One of the best ways to ensure that the software product will operate as defined by the *requirements* is to run *end-to-end scenarios* with validation. This requires one to have a list of *functional specifications* that the software product must perform, and to create one or more workflows where these will be pipelined together to generate this type of *Functional Testing*[4].

For BMI these are defined as follows:

- `pro` ► *Provisions a node.*
- `dpro` ► *Deprovisions a node.*

[2] Coverage ensures all aspects of the codebase are verified and asserted via test(s).

[3] Ashfaque Ahmed and Bhanu Prasad. 2016. *Foundations of Software Engineering.* Auerbach Publications, Boston, MA, USA.

[4] Functional Testing validates the software design based on the requirement specifications, by running tests to check that the software's features match the functional specifications.

- `snap` ▶ *Takes a snapshot of a node.*
- `ls` ▶ *Lists store images.*
- `import` ▶ *Importing images or snapshots into BMI for provisioining.*
- `db` ▶ *Database commands that about imported images or snapshots.*

By then integrating these into an end-to-end workflow, one can perform all these and ensure that the basic requirements are satisfied. An example of a possible end-to-end workflow is described in Figure 1.2.

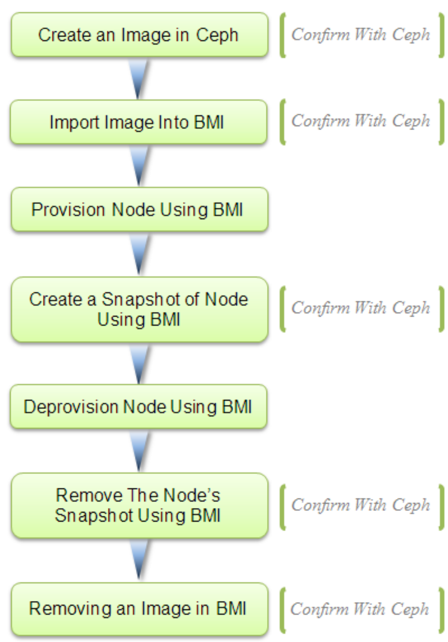


FIGURE 1.2 – An example of an end-to-end workflow.

1.3 BEHAVIOR-DRIVEN DEVELOPMENT: A SCIENTIFIC-METHOD APPROACH TO TESTING

The *Scientific Method* is an unbiased approach to discovering what the facts truly about a system by progressing using *systematic doubt*[5] to ensure adequate evidence support each problem being solved. The facts are not gathered unless there is a problem being defined upon which relevant facts are required to prove or disprove inquiries (*hypotheses*) about the problem.

[5] Morris Raphael Cohen and Ernest Nagel. 1934. *An Introduction to Logic And Scientific Method.* Harcourt, Brace and World, New York, NY, USA.

Behavior-Driven Development (BDD) is defined through a live document implemented using the Gherkin language[6], which utilizes *Given-When-Then* control-flow syntax defined as follows:

`Given` ▶ *Defines a given state.*
`When` ▶ *Defines a given action performed under the given state.*
`Then` ▶ *Defines the expected outcome after the action is performed.*

By building a *scenario* through combining these into premises using the *Given-* and *When-*initiated statements, we are able to discover if our system is validated at each step and confirm the *Then* conclusion statement(s). Thus we are hypothesis-driven through a BDD-model of our system to ensure it matches our expected *operational semantics*[7].

An example of such an end-to-end scenario for BMI is illustrated in Figure 1.3, where each line is a step that references an implemented function.

The end-to-end scenario is a *model* used as a set of *rules of inference* guided by an ordered collection of *premises* – which are assumed to be *true* – and conclude that *all* the steps are *truth preserving*:

$$Premises \implies Conclusions$$

[6] https://github.com/cucumber/cucumber/wiki/Gherkin
[7] https://en.wikipedia.org/wiki/Operational_semantics

```
Feature: Running an end-to-end acceptance test

  Scenario: Importing/Removing Image, DB/Ceph consistency

    Given RBD will create an image
        | image_name     |
        | bmi-test-image |

    And BMI log line-count will be measured at the beginning

    When BMI will import an image
        | image_name     | project_name  |
        | bmi-test-image | bmi_infra     |

    And BMI will provision a node
        | image_name     | project_name  | network_name      | node_name  |  NIC       |
        | bmi-test-image | bmi_infra     | bmi-provision-dev | cisco-05   | enp130s0f0 |

    Then BMI will create a snapshot of a node
        | project_name  | node_name      |  snapshot_name           |
        | bmi_infra     | cisco-05       |  bmi-test-image-snapshot |

    Then RBD will confirm the snapshot exists
        | snapshot_name           |
        | bmi-test-image-snapshot |

    And BMI will remove a snapshot
        | snapshot_name           | project_name  |
        | bmi-test-image-snapshot | bmi_infra     |

    Then BMI will deprovision a node
        | project_name  | network_name      | node_name  |  NIC       |
        | bmi_infra     | bmi-provision-dev | cisco-05   | enp130s0f0 |

    And BMI will remove an image
        | image_name     | project_name  |
        | bmi-test-image | bmi_infra     |

    Then RBD will confirm the removed image's clone
        | image_name     | project_name  |
        | bmi-test-image | bmi_infra     |

    And RBD will remove the created image
        | image_name     |
        | bmi-test-image |

    And BMI log line-count will be measured at the end
```

FIGURE 1.3 – The BMI End-to-End Behavior-Driven Deployment Test, with tables of parameters to test with.

For example, in Figure 1.4 the creation of a RADOS block device (RBD[8]) mountable image at the start is defined through the `rbd_create_image()` function, where it is decorated by the sentence referenced in the live-document.

[8] Ceph Storage provides the ability for its (bootable) images to be mountable remotely using a RADOS block device. For more information please proceed to the following web location: https://docs.openstack.org/mitaka/config-reference/block-storage/drivers/ceph-rbd-volume-driver.html

```
import behave
import time # Needed for Ceph Hammer client consistency
from bmi_config import RBD_CREATE, IMAGE_NAME, PROVISIONING_DELAY
from subprocess import check_output, CalledProcessError, STDOUT
from test_operation import test_event_store_insert, test_rollback

@step('RBD will create an image')
def rbd_create_image(context):
    for row in context.table:
        try:
            print( "      -> Checking that no pre-existing " +
                                 row['image_name'] + " is present in Ceph, before creating it...")
            rbd_filename_check_stdout = check_output('rbd ls | grep ' + row['image_name'],
                                                                stderr=STDOUT, shell=True)
        except CalledProcessError:
            pass # The image already exists, as it was previously created

        try:
            print( "      -> Creating the " + row['image_name'] + " image in Ceph...")
            rbd_create_stdout = check_output( 'rbd create ' +
                                               row['image_name'] +
                                              ' --size 1 --image-format 2', stderr=STDOUT, shell=True)
        except CalledProcessError:
            pass # The image already exists, as it was previously created
        except Exception:
            test_rollback(context)
        print( "      -> Checking that " + row['image_name'] + " exists in Ceph...")
        rbd_filename_check_stdout = check_output( 'rbd ls | grep ' +
                                                   row['image_name'], stderr=STDOUT, shell=True)
        context.rbd_filename_check = rbd_filename_check_stdout.strip()

        # Journal the event for rollback
        test_event_store_insert(context, { RBD_CREATE: {IMAGE_NAME : 'image_name'} })

        time.sleep( PROVISIONING_DELAY ) # Needed for Ceph Hammer client consistency
        assert context.rbd_filename_check == row['image_name']
```

FIGURE 1.4 – The definition of the RBD creation step, where the decoration highlights the sentence referenced in the end-to-end deployment test.

In the next chapter, you will learn how to configure and run an acceptance test.

II

Getting Started With The User Acceptance Testing Framework

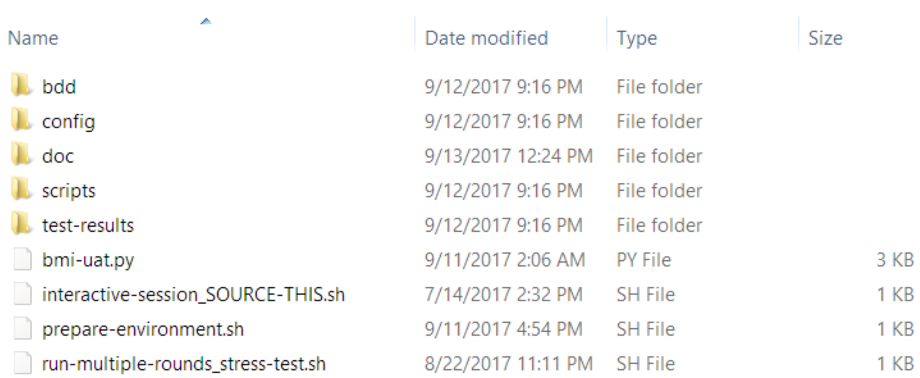
CHAPTER 2

The User Acceptance Testing Framework

This chapter will guide through the steps of creating and running a BDD scenario for BMI via the User Acceptance Framework.

2.1 The User Acceptance Testing Architecture

You will need to be provided a compressed (*tar.gz*) file of the acceptance tests. After you uncompress it via the "`tar -xzvf acceptance-tests.tar.gz`" you will see the following files and folders in the root directory:



Name	Date modified	Type	Size
bdd	9/12/2017 9:16 PM	File folder	
config	9/12/2017 9:16 PM	File folder	
doc	9/13/2017 12:24 PM	File folder	
scripts	9/12/2017 9:16 PM	File folder	
test-results	9/12/2017 9:16 PM	File folder	
bmi-uat.py	9/11/2017 2:06 AM	PY File	3 KB
interactive-session_SOURCE-THIS.sh	7/14/2017 2:32 PM	SH File	1 KB
prepare-environment.sh	9/11/2017 4:54 PM	SH File	1 KB
run-multiple-rounds_stress-test.sh	8/22/2017 11:11 PM	SH File	1 KB

Figure 2.1 – Root directory of the User Acceptance Testing framework.

Below are descriptions of the critical folders and files required for configuring and for running the tests:

- `config` ▶ *Contains the testing configurations for different environments (i.e. PRB, NEU, etc).*
- `doc` ▶ *Contains this manual.*
- `scripts` ▶ *Contains the configuration scripts that are run for different stages during testing.*
- `prepare-environment.sh` ▶ *Configurations to run for different OS environments before testing, which can be used to create cleanup scripts.*
- `bmi-uat.py` ▶ *The command-line interface (CLI) for listing and running the tests.*
- `test-results` ▶ *Provides test results in case one performs randomized tests for multiple rounds.*

The `interactive-session_SOURCE-THIS.sh` script is used if you want to drop into an interactive session into the environment of a particular test after is completed in order to inspect or rollback changes.

Next you will learn about the `config` directory regarding how to use or create new testing environments.

2.1.1 *Configuring a Testing Environment*

If you look at the `config` directory you will see something that looks similar to this:

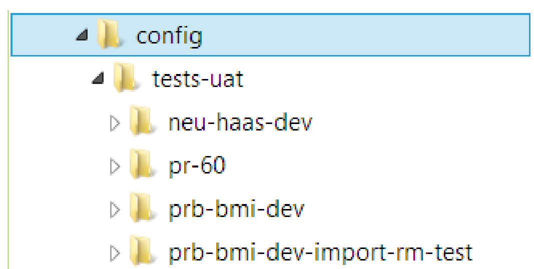


FIGURE 2.2 – `Config` directory of the User Acceptance Testing framework.

It is best to copy a previous directory of interest if you would like to perform the minimal changes to a test. There are two types of test directories:

- `Pull-Request Test` ▶ *Performs tests on a specific pull-request (i.e.* pr-60*). These are usually performed in preparation for running a deployment test.*
- `Repository Test` ▶ *Performs tests on a whole repository (i.e.* neu-haas-dev*). These would be performed to ensure a release is ready for deployment.*

Next we will look at how a test configuration is structured.

2.1.2 *Structure of Test Directory*

If you look at any of the test directories they all look as follows:

Name	Date modified	Type	Size
doc	9/12/2017 9:16 PM	File folder	
features	9/12/2017 9:16 PM	File folder	
scripts	9/12/2017 9:16 PM	File folder	
steps	9/12/2017 9:16 PM	File folder	
bmi-config.sh	7/14/2017 6:41 PM	SH File	1 KB
config	9/11/2017 2:04 AM	File	1 KB
customize	7/14/2017 10:09 PM	File	1 KB
customize-after-git-clone	7/14/2017 5:57 PM	File	1 KB

FIGURE 2.3 – The test directory structure.

To keep the configurations simple and practical, it is important to know about the following four components:

`config` ▶ *This file configures the BMI UAT test for the environment, and is the most important file.*

`bmi-config.sh` ▶ *This is the second most important file, and is used to configure the BMI pre-test deployment directories.*

`features` ▶ *These contain the live documents that can be changed with the exception of the* template *file. Additional scenario files to test for can be added if preferred.*

`steps` ▶ *Contains the functions that map to the given BDD definition in the* feature *files that build up the scenarios.*

2.1.3 *The* `config` *file*

The `config` file is usually the only file one will usually configure the most of the time, and it was created to ensure minimal changes are necessary for test-preparation. The structure of the file is shown in Figure 2.4, and is composed of the following three main sections:

`BMI_RELEASE_NAME` ▶ *This will denote the name of the directory for the scenario that is being tested, underneath which the tests will be installed, configured and run.*

`E2E Test Configs` ▶ *The middle section contains the End-To-End configuration information that are pertinent to the environment being tested (i.e. HIL project names, names of BMI images to create, etc).*

`BMI and HIL Configs` ▶ *These contain the HIL and BMI local configurations in order for the tests to run.*

```
export BMI_RELEASE_NAME=moc-0.5-release

export BMI_PROJECT=bmi_infra
export HIL_NODE=cisco-05
export HIL_NIC=enp130s0f0
export HIL_NETWORK=bmi-provision-dev
export BMI_IMAGE_NAME=bmi-test-image
export BMI_SNAPSHOT_NAME=bmi-test-image-snapshot

export BMI_CONFIG=/etc/bmi/bmiconfig_pgrosu.cfg
export HIL_ENDPOINT=http://127.0.0.1:8000
export HIL_USERNAME=*********
export HIL_PASSWORD=*********
```

Figure 2.4 – The test configuration file structure.

An example of the `bmi-config.sh` file is shown in Figure 2.4, which provides the configurations of where the BMI instance will be installed (`BMI_INSTANCE_DIR`), and the location of the User Acceptance Tests directory (`ACCEPTANCE_TESTS_SRC_DIR`).

```
export BMI_INSTANCE_DIR=${HOME}/pgrosu/ims-instance
export ACCEPTANCE_TESTS_SRC_DIR=${HOME}/pgrosu/acceptance-tests
export BDD_DIR=./bdd
export BDD_STEPS_DIR=$BDD_DIR/steps
```

Figure 2.5 – An example of a `bmi-config.sh` file.

2.2 PREPARING THE ENVIRONMENT

Sometimes the operating environment requires extra functionality – such as Python or Git availability – to be available before running a test. These configurations can be placed as Bash scripts under the `scripts\prepare-environments` directory, as shown in Figure 2.6.

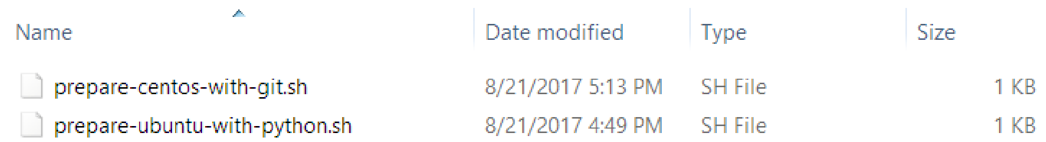


FIGURE 2.6 – An example of the `prepare-environments` directory.

To list all configurations, type under the main `acceptance-tests` directory `./prepare-environment.sh`, as shown in Figure 2.9.

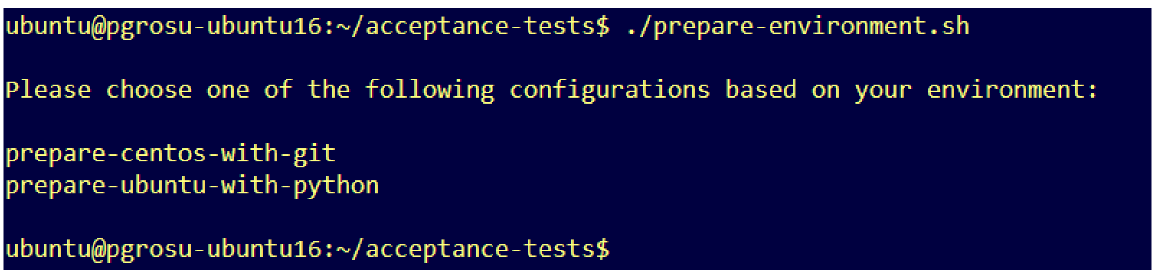


FIGURE 2.7 – Listing the `prepare-environments` configurations.

To run a configuration just use the following format to run the appropriate configuration for your environment:

```
./prepare-environment.sh CONFIGURATION
```

Now you are ready to run a test configuration.

2.3 PERFORMING THE ACCEPTANCE TESTS

The performance tests can be initiated via the following steps:

1. To list the testable BMI service configurations, type the following command:

   ```
   ./bmi-uat.py ls
   ```

   You should see something like the following:

   ```
   The available configurations are:
   ```

```
neu-haas-dev
```

2. To run the standard end-to-end configuration, type the following command:

```
./bmi-uat.py --run BMI_SERVICE_CONFIGURATION
```

Example:

```
./bmi-uat.py --run neu-haas-dev
```

At the end you if the tests passed successfully, you should see the following output:

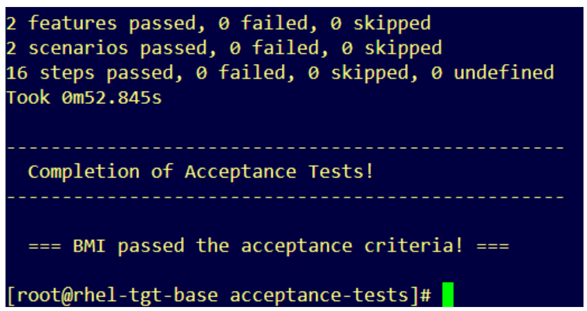


Figure 2.8 – The BMI completed successfully the end-to-end scenario.

3. To run the tests with randomized parameters, type the following where the value indicates the number of times to run the test:

```
./bmi-uat.py --run neu-haas-dev --randomize 3
```

4. To check if the tests passed or failed, type the following:

```
./bmi-uat.py check
```

You should see the following:

```
All tests passed!
```

This command checks the `test-results` directory for any subdirectory containing `FAIL` in its name.

5. To cleanup all previous results, type the following:

```
./bmi-uat.py clean
```

You will notice that when running a test, there are many additional sanity-checks that are being made to ensure each test not only completes properly, but also provides sufficient detail in case of failure, as shown by the following figure:

```
  When BMI will import an image                               # tests/bdd/steps/bmi_import.py:7
    | image_name     | project_name |
    | bmi-test-image | bmi_infra    |
       Running: bmi import bmi_infra bmi-test-image
    -> Checking that bmi-test-image exists in Ceph...
       Running: bmi db ls | grep bmi-test-image
    -> Checking that 112233445566778899img2 exists in BMI's database...
       Running: rbd ls | grep 112233445566778899img2
```

FIGURE 2.9 – Sanity-checks for the BMI import step.

2.3.1 *Entering an Interactive Session*

After the test has completed – either successfully or not – one can enter an interactive session to inspect the state of the test, where BMI commands can be executed interactively. This is performed from the `acceptance-tests` directory by typing the following command – make sure to not forget the period (.) at the before the script-name:

```
. interactive-session_SOURCE-THIS.sh
```

If the test was based on a pull-request, please add -pr as follows:

```
. interactive-session_SOURCE-THIS.sh -pr
```

You should see something as follows, showing that one is an virtual environment:

```
(.bmi_venv) ubuntu@pgrosu-ubuntu16:~/ims-instance/ims$
```

To exit the interactive session, just type the following command – from within the session:

```
. return-back-to-acceptance-dir_SOURCE-THIS.sh
```

By following the above steps you can now test your own customizations of BMI any services.

Index