# The OVS Integration Test suite (TOIT) High Level Design

Document Number: 1.0

Document Revision: 0.2

Date: April 2015

## Copyright Notice

| Document Owner | First Name | Last Name | |
|---|---|---|---|
| | Maryam | Tahhan | maryam.tahhan@intel.com |

| Primary Authors | First Name | Last Name | |
|---|---|---|---|
| | Maryam | Tahhan | maryam.tahhan@intel.com |
| | Meghan | Halton | meghan.halton@intel.com |
| | Billy | O Mahony | billy.o.mahony@intel.com |

| Key Contributors | First Name | Last Name | |
|---|---|---|---|
| | Stephen | Finucane | stephen.finucane@intel.com |
| | Michal | Weglicki | michalx.weglicki@intel.com |
| | Timo | Puha | timox.puha@intel.com |

## Approvals

| Name | Email | Date |
|---|---|---|
| | | |
| | | |

# Abstract

The OVS Integration Test suite High Level Design document presents the test framework that is being put forward for Project "vSwitch Performance Characterization".

The main goal of VSPERF is to develop a generic and architecture agnostic vSwitch testing framework and associated tests that will serve as a basis for validating the suitability of different vSwitch implementations in Telco NFV environments.

TOIT was extended as a starting point to help achieve the VSPERF project goal. However, TOIT's current architecture doesn't expose generic enough APIs to test cases to allow for different vSwitches and traffic generators to be plugged in without test cases knowing the internals of their implementation details. As well as this, TOIT in its current format does not meet the agreed upon directory structure of the VSPERF project in OPNFV, thus **TOIT needs to be redesigned to fulfil these two requirements.**

This document will provide an overview of the current architecture of TOIT and will present an alternative architecture that will address the main concerns for redesign.

## Revision History

| Revision | Description | Date | Author |
|---|---|---|---|
| 0.1 | Initial Version | 09/03/2015 | Maryam Tahhan<br>Meghan Halton |
| 0.2 | Detailed Revision | 18/04/2015 | Maryam Tahhan<br>Meghan Halton<br>Billy O Mahony |
| | | | |

## Table of Contents

# 1. Introduction

## 1.1 Terminology

The table below presents the definition of terms used in this document.

*Table 1: Terms and definitions*

| Term | Description |
|---|---|
| TOIT | The vSwitch Integration/Performance Test suite |
| vSwitch | Virtual Switch |
| OVS | Open vSwitch |
| VSPERF | Characterize vSwitch Performance for Telco NFV Use Cases |
| VNF | Virtual Network Function |

## 1.2 References

*[0] Chapter 3: Architectural Patterns and Styles, Component Based Style [online], available:* https://msdn.microsoft.com/en-us/library/ee658117.aspx *[accessed March 2015].*

[1] Component: Game Programming Patterns/ Decoupling Patterns [online], available: http://gameprogrammingpatterns.com/component.html [accessed March 2015].

[2] What are the advantages and disadvantages of a layered architecture [online], available: http://murlid05.blogspot.com/2012/06/what-are-advantages-and-disadvantages.html *[*accessed Mar 2015*].*

[3] Clements, P., Kazaman, R., Klein, M. (2004) 'Evaluating Software Architectures: Methods and Case Studies', Addison-Wesley.

[4] 'The Architecture Trade-off Analysis Method' [online], available: http://www.sei.cmu.edu/architecture/ata_method.html [accessed Mar 2015].

[5] Zayaraz, G., Thambidurai, P. (2005) 'Software Architecture Selection Framework Based on Quality Attributes' [online], available: http://ieeexplore.ieee.org/xpls/icp.jsp?arnumber=1590147 [accessed Mar 2015].

[6]Giesen, J., VÖlker, A. (2002) 'Requirements Interdependencies and Stakeholder's Preferences' [online], available: http://ieeexplore.ieee.org/xpls/icp.jsp?arnumber=1048528 [accessed Mar 2015].

[7] Cheesman, J., Daniels, J. (2001) ‘UML Components: A Simple Process for Specifying Component-Based Software’, Addison-Wesley.

# 2. Assumptions and Dependencies

## 2.1 Assumptions

- TOIT provides suitable functionality in order to be able to integrate the performance tests set out by the VSPERF test specification.
- TOIT will expose generic APIs for test components such as vSwitches, traffic generators, and VNFs to allow for the integration with multiple component implementations.
- TOIT will conform to the agreed upon directory structure for the “Characterize vSwitch Performance for Telco NFV Use Cases” OPNFV project.
- OVS is enabled with a DPDK guest access method, ideally vhost.

## 2.2 Dependencies

- “Characterize vSwitch Performance for Telco NFV Use Cases” Test Specification
- Existing TOIT
- DPDK
- OVS
- QEMU version < 2.0 or > 2.0 selected by OPNFV for Release 1.
- IXIA
- Python 3.x version supported by CentOS 7.
- CentOS 7.

# 3. TOIT Architecture and Design

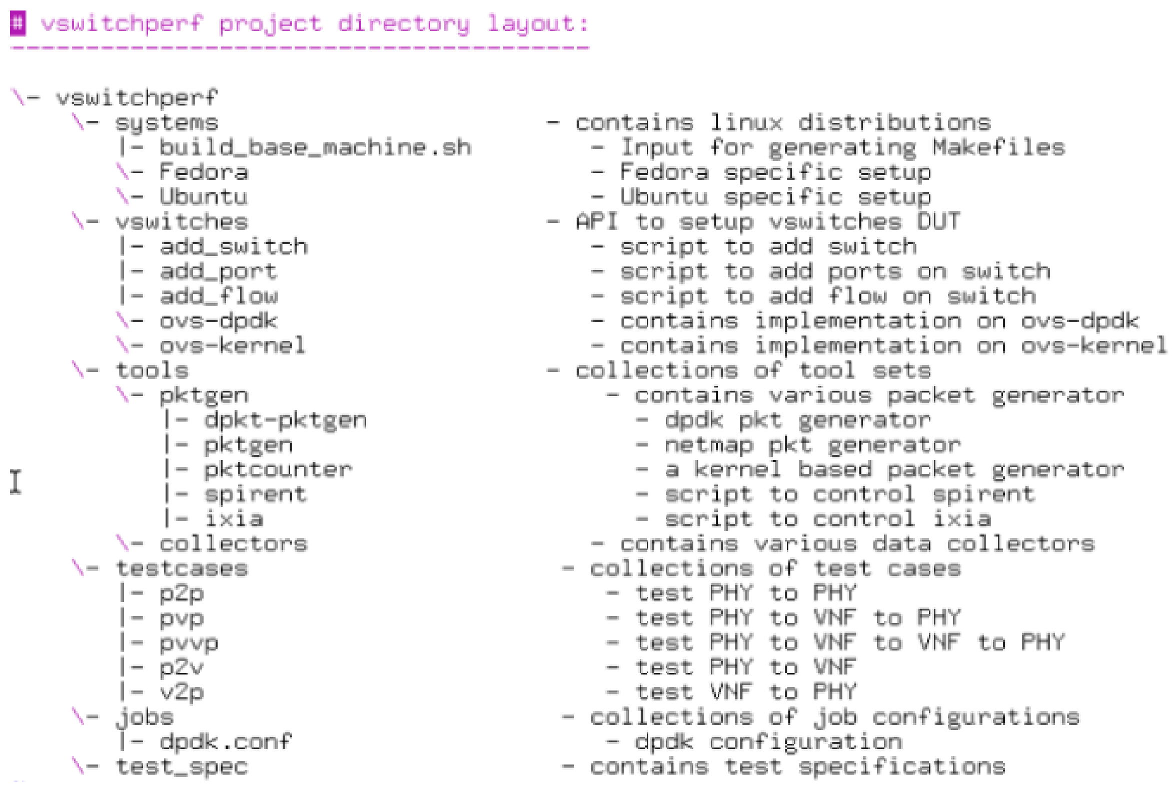
## 3.1 Aim

TOIT needs to be redesigned to expose more generic APIs for low level test components, such as a virtual switch, to higher level test case implementations. It also must be restructured to fit into the predefined repository structure for the “Characterize vSwitch Performance for Telco NFV Use Cases” OPNFV project.

## 3.2 Overview

TOIT is an Apache licensed test framework that was developed in Python 2.7 to enable integration testing of Open vSwitch (OVS), and in particular the netdev-dpdk bridge in OVS. TOIT was adopted for the the “Characterize vSwitch Performance for Telco NFV Use Cases” OPNFV project (VSPERF). The main goal of VSPERF is to develop a generic and architecture agnostic vSwitch testing framework and associated tests that will serve as a basis for validating the suitability of different vSwitch implementations in Telco NFV environments. The output of this project will be utilized by the OPNFV Performance and Test group and its associated projects, as part of OPNFV Platform and VNF level testing and validation.

TOIT was extended as a starting point to help achieve the project goal. However, TOIT’s current format is not generic enough to cater for different vSwitches and does not meet the agreed upon directory structure of the VSPERF project in OPNFV, thus **TOIT needs to be redesigned to be more generic and to allow it to fit into the predefined repository structure represented in Figure 1.**



```
# vswitchperf project directory layout:
---------------------------------------

\- vswitchperf
   \- systems                        - contains linux distributions
      |- build_base_machine.sh          - Input for generating Makefiles
      \- Fedora                         - Fedora specific setup
      \- Ubuntu                         - Ubuntu specific setup
   \- vswitches                      - API to setup vswitches DUT
      |- add_switch                     - script to add switch
      |- add_port                       - script to add ports on switch
      |- add_flow                       - script to add flow on switch
      \- ovs-dpdk                       - contains implementation on ovs-dpdk
      \- ovs-kernel                     - contains implementation on ovs-kernel
   \- tools                          - collections of tool sets
      \- pktgen                          - contains various packet generator
         |- dpkt-pktgen                    - dpdk pkt generator
         |- pktgen                         - netmap pkt generator
         |- pktcounter                     - a kernel based packet generator
         |- spirent                        - script to control spirent
         |- ixia                           - script to control ixia
      \- collectors                     - contains various data collectors
   \- testcases                       - collections of test cases
      |- p2p                             - test PHY to PHY
      |- pvp                             - test PHY to VNF to PHY
      |- pvvp                            - test PHY to VNF to VNF to PHY
      |- p2v                             - test PHY to VNF
      |- v2p                             - test VNF to PHY
   \- jobs                            - collections of job configurations
      |- dpdk.conf                       - dpdk configuration
   \- test_spec                       - contains test specifications
```

Figure 1: Predefined VSPERF directory Layout

## 3.3 Current Architecture and Design

Before walking through the redesign one must first understand the current design of TOIT.

TOIT has 6 main modules, these are:

- **guest**: Implements control of the VMs. Currently only QEMU/KVM is supported.
- **ovs**: Control of Open vSwitch, including addition and removal of bridges, ports, flows and control of the vSwitch daemon using the ovs-vsctl, ofctl and vSwitchd commands.
- **sysmetrics**: Implements system metrics logging. Currently only the python module linux-metrics is supported. Implemented in a modular way enabling other metrics loggers to be swapped in easily.
- **system:** Manages configuration of the system (Linux, DPDK etc.) for Open vSwitch.
- **test:** Manages the loading and listing of tests.
- **trafficgen:** Manages the sending and receiving of traffic in different modes. Currently supports the following modes:
    - Burst
    - Continuous
    - RFC2544 Throughput
    - RFC2544 Back2Back

  Implemented in a modular way enabling other traffic generators to be swapped in easily. Currently Supports:

    - IxNet

- IxExplorer (Called Ixia)
- Dummy

TOIT also provides two "helper" modules

- **conf**: Manages the configuration of TOIT (Paths, trafficgen parameters, expected results etc.)
- **utils:** General purpose utilities such as task management, plugin loader, and short term result storage.

### 3.3.1 Architecture and Design Diagrams

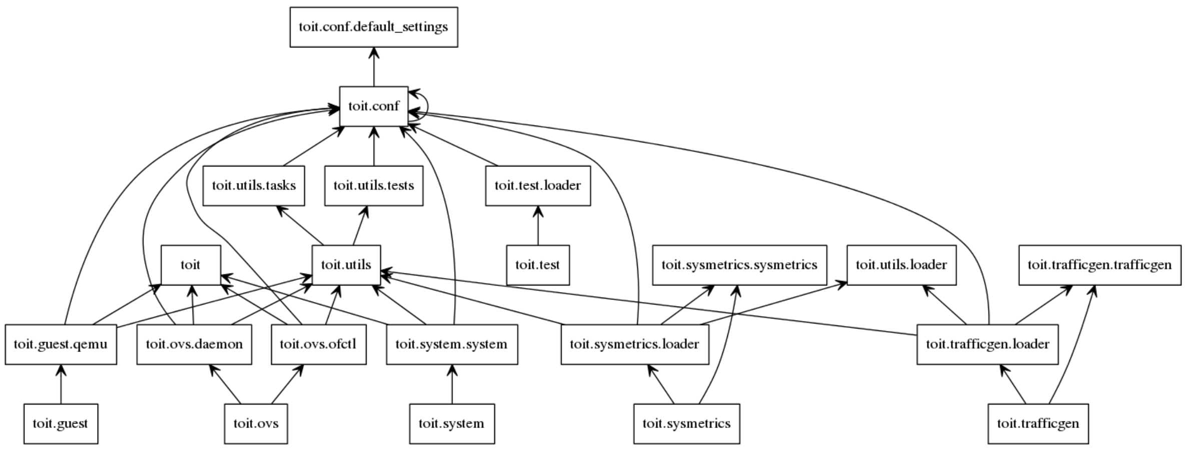


*Figure 2: Package diagram of TOIT before rework*

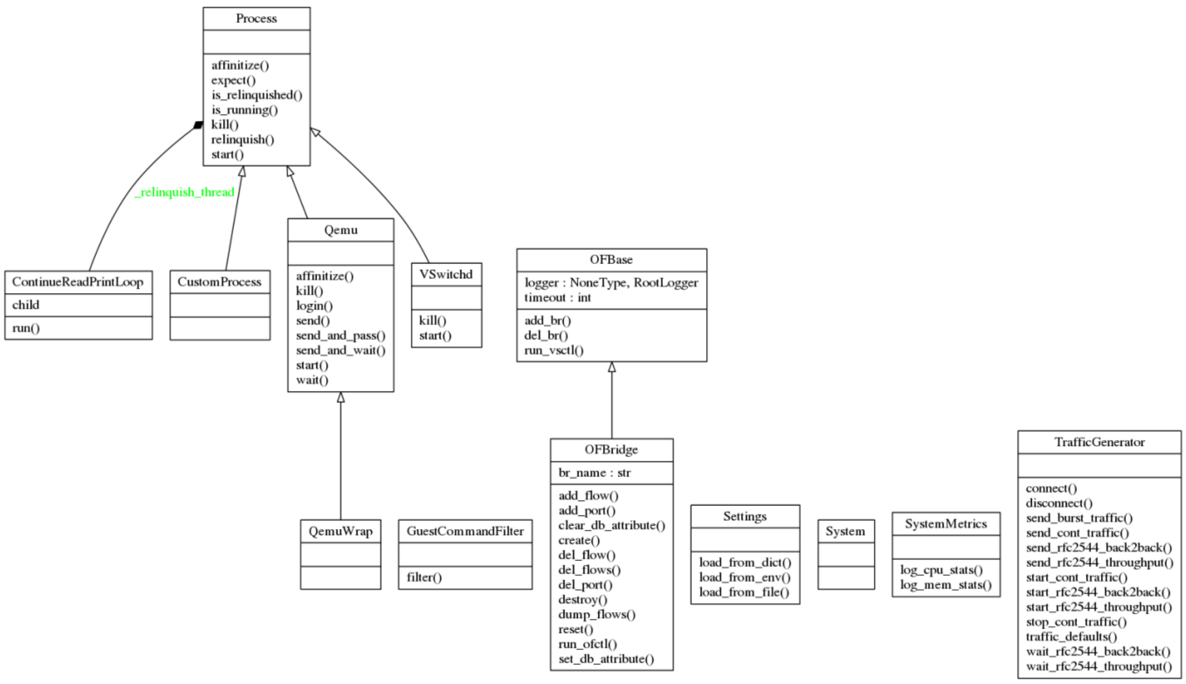
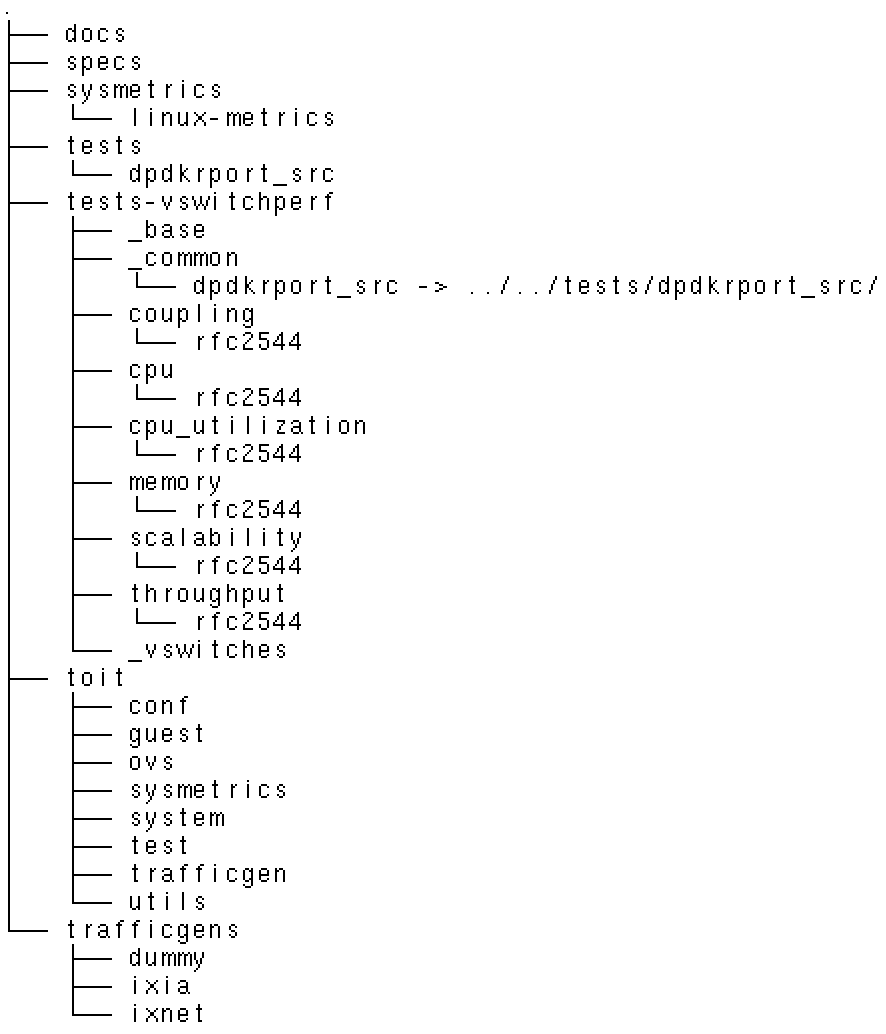


*Figure 3: Class diagram of TOIT before rework*



```
.
├── docs
├── specs
├── sysmetrics
│   └── linux-metrics
├── tests
│   └── dpdkrport_src
├── tests-vswitchperf
│   ├── _base
│   ├── _common
│   │   └── dpdkrport_src -> ../../tests/dpdkrport_src/
│   ├── coupling
│   │   └── rfc2544
│   ├── cpu
│   │   └── rfc2544
│   ├── cpu_utilization
│   │   └── rfc2544
│   ├── memory
│   │   └── rfc2544
│   ├── scalability
│   │   └── rfc2544
│   ├── throughput
│   │   └── rfc2544
│   └── _vswitches
├── toit
│   ├── conf
│   ├── guest
│   ├── ovs
│   ├── sysmetrics
│   ├── system
│   ├── test
│   ├── trafficgen
│   └── utils
└── trafficgens
    ├── dummy
    ├── ixia
    └── ixnet
```

*Figure 4: Current TOIT directory layout*

## 3.4 TOIT Re-Architecture

### 3.4.1 Introduction

As a TOIT architecture design document did not exists for the previous implementation of TOIT, a methodology called Architecture Trade-off Analysis Method (ATAM) will be applied in order to identify the stakeholders, scope, functionality and the quality attributes of TOIT.

### 3.4.2 Architecture Trade-off Analysis Method (ATAM)

Architecture Trade-off Analysis Method (ATAM) is a method for the identification and evaluation of a software architecture. This method provides scope for the identification, documentation and evaluation of the architecture of TOIT, particularly how well the suggested architecture meets TOIT's quality attributes [3], as well as the identification of any possible risks early in the lifecycle. Generally it involves an evaluation team, architects and project stakeholders [4]. For the application of this method, the authors, as well as the VSPERF committers and contributors will assume these roles as needs be. For more information about ATAM please see Appendix A.

#### *3.4.2.1 TOIT Presentation: Business drivers and Architecture*

A system overview will be used to identify TOIT's business drivers. This overview will include: TOIT's main functionalities, constraints, major stakeholders and the architectural drivers [3].

*TOIT Overview and Constraints*

TOIT a generic and architecture (hardware) agnostic vSwitch testing framework. It will serve as a basis for validating the suitability of different vSwitch implementations in Telco NFV environments. The Characterize vSwitch Performance for Telco NFV Use Cases" OPNFV project provides a test specification that describes the suite of tests to characterize the performance of a virtual switch for Telco NFV use cases. TOIT is a realization of the specified test cases.

As part of the testing procedure TOIT is required to configure and setup any dependencies that a supported vSwitch requires for a test. As well as this, TOIT must report the results of any tests that are run. Finally TOIT will be easily extensible to support new Tests, Traffic Generators, vSwitches, and VNFs

*System Main Functionalities*

TOIT users will fall under 2 categories: testers and developers. TOIT needs to allow a tester to do the following:

- Configure TOIT.
- Run a test/suite of tests:
    - Setup a vSwitch and any of its dependencies.

        - Run the actual test.
        - Report results.
    - Print settings.

It also needs to allow a user to do the following:

- Add a test, vSwitch, traffic generator or a VNF.
- It should produce XML (xUnit) style output for CI servers.
- It should export shell scripts that can be used in place of the test framework.

## Technology

**Programming Languages**: The original chosen language for TOIT was Python 2.7 (LTS, released in July 2010) However as part of the re-implementation TOIT will be upgraded to use Python 3.x (that is supported by CentOS 7). TCL scripting will also be used to configure and use the IXIA traffic generator. Other implementations of traffic generators, VNFs, vSwitches or collectors which are added later on may require control via languages other than those mentioned.

**Documentation Languages:** Markdown is the documentation language of choice as it provides a way to implement hyperlinking between documents/sections while retaining readability when viewed in a plain text editor.

**External Modules:** TOIT makes use of the Jinja module to define and create test reports as well as Pylint and Tox modules to ensure code quality.

**Targeted OS:** CentOS 7 and Ubuntu 14.04

**Targeted QEMU:** < 2.0 or > 2.0 (depends on CentOS 7 support).

### *TOIT Major Stakeholders*

In order to identify the primary stakeholders the concept termed "segmentation" was applied. Segmentation refers to the grouping of people based on similar characteristics [5]. The characteristic that is most likely to be linked to a person is their job responsibility/role [6]. Thus the main stakeholders are: the user (tester/developer), the VSPERF committers and contributors and the software architects (the authors).

### *TOIT Architectural Drivers*

This section identifies the major quality attributes that affect the architecture [3]. The architectural drivers that are central to the success of the proposed system include:

- **Modifiability, Modularity and Extensibility:** It is central to the system's success that other tests, traffic generators, virtual machine managers and virtual switches can be added to the system in the future. They should also be easy to add; as such each of these components will need to be relatively modular.

## *TOIT System Architecture:*

This section will provide an overview of the system architecture including: technical constraints, other software with which the system must interact, system diagrams, identification of architectural patterns, and the Architectural approaches and decisions.

### Technical Constraints

TOIT is subject to some constraints including:

- As much as possible of the original TOIT should be re-used.
- The TOIT redesign must fit into the agreed upon directory structure specified by the VSPERF project.
- TOIT must be modular and easily extensible.

The tests implemented in TOIT must include or implement the following parameters unless explicitly stated by the test case that it does not support the specific parameters:

- Frame size (bytes): 64, 128, 256, 512, 1024, 1280, 1518, 2K, 4k OR Packet size based on use-case (e.g. RTP 64B, 256B).
- Reordering check: Tests should confirm that packets within a flow are not reordered.
- Duplex: Unidirectional / Bidirectional. Default: Full duplex with traffic transmitting in both directions, as network traffic generally does not flow in a single direction. By default the data rate of transmitted traffic should be the same in both directions, please note that asymmetric traffic (e.g. downlink-heavy) tests will be mentioned explicitly for the relevant test cases.
- Number of Flows: Default for non-scalability tests is a single flow. For scalability tests the goal is to test with maximum supported flows but where possible will test up to 10 Million flows. Start with a single flow and scale up. By default flows should be added sequentially, tests that add flows simultaneously will explicitly call out their flow addition behaviour. Packets are generated across the flows uniformly with no burstiness.
- Traffic Types: UDP, SCTP, RTP, GTP and UDP traffic.
- Deployment scenarios are:
  - Physical → virtual switch → physical.
  - Physical → virtual switch → VNF → virtual switch → physical.
  - Physical → virtual switch → VNF → virtual switch → VNF → virtual switch → physical.
  - Physical → virtual switch → VNF.
  - VNF → virtual switch → Physical.
  - VNF → virtual switch → VNF.

### Other Software or systems with which TOIT must interact

It is expected that TOIT will interact with several libraries and tools that could be categorised under: vSwitches, VNFs, hypervisors, traffic generators and collectors.

The scope of the first set of TOIT patches to VSPERF will ensure that TOIT interacts with the following system tools and libraries:

- Open vSwitch (accelerated with DPDK)
- DPDK
- IXIA and a dummy vSwitch.

Other system utilities TOIT might interact with are:

- Qemu
- Memtester
- Stress

The targeted test case for the first set of TOIT patches to VSPERF is: LTD.Throughput.RFC2544.PacketLossRatio

## System Diagrams

This section includes a structural and development view of the TOIT architecture. The structural view (cf. Figure 5) presents a high level overview of the proposed components for TOIT. The development view (cf. Figure 6) shows the packages that compose the various components; the internal structure of these packages will be discussed in section 3.5.

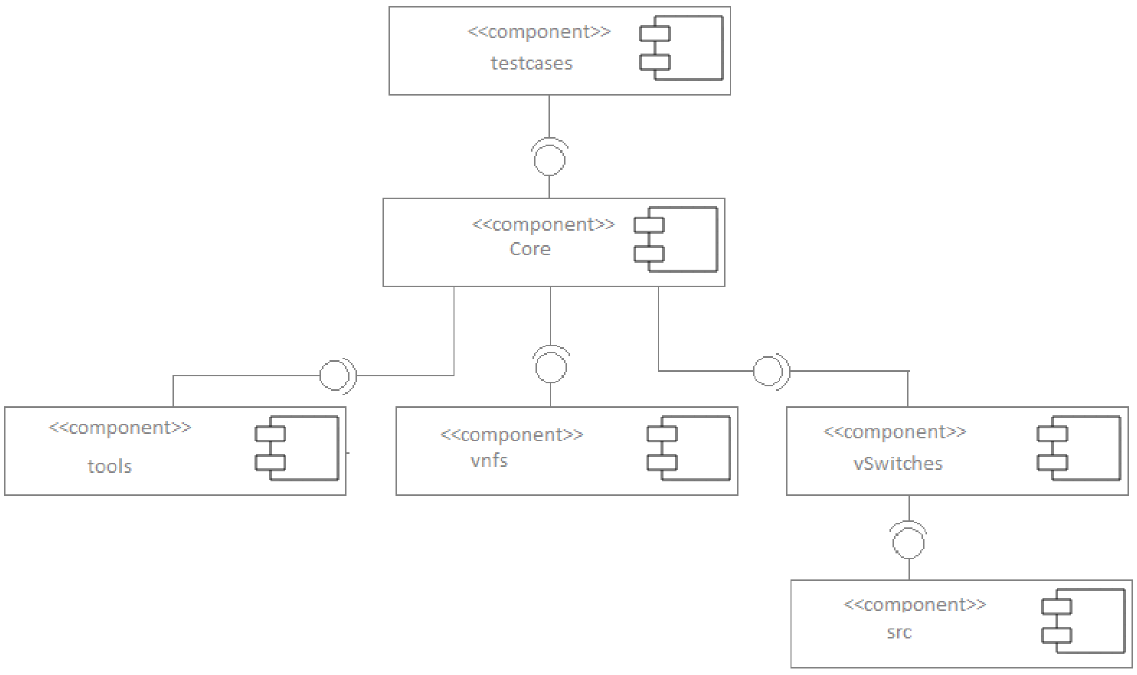


*Figure 5 Structural view of TOIT*

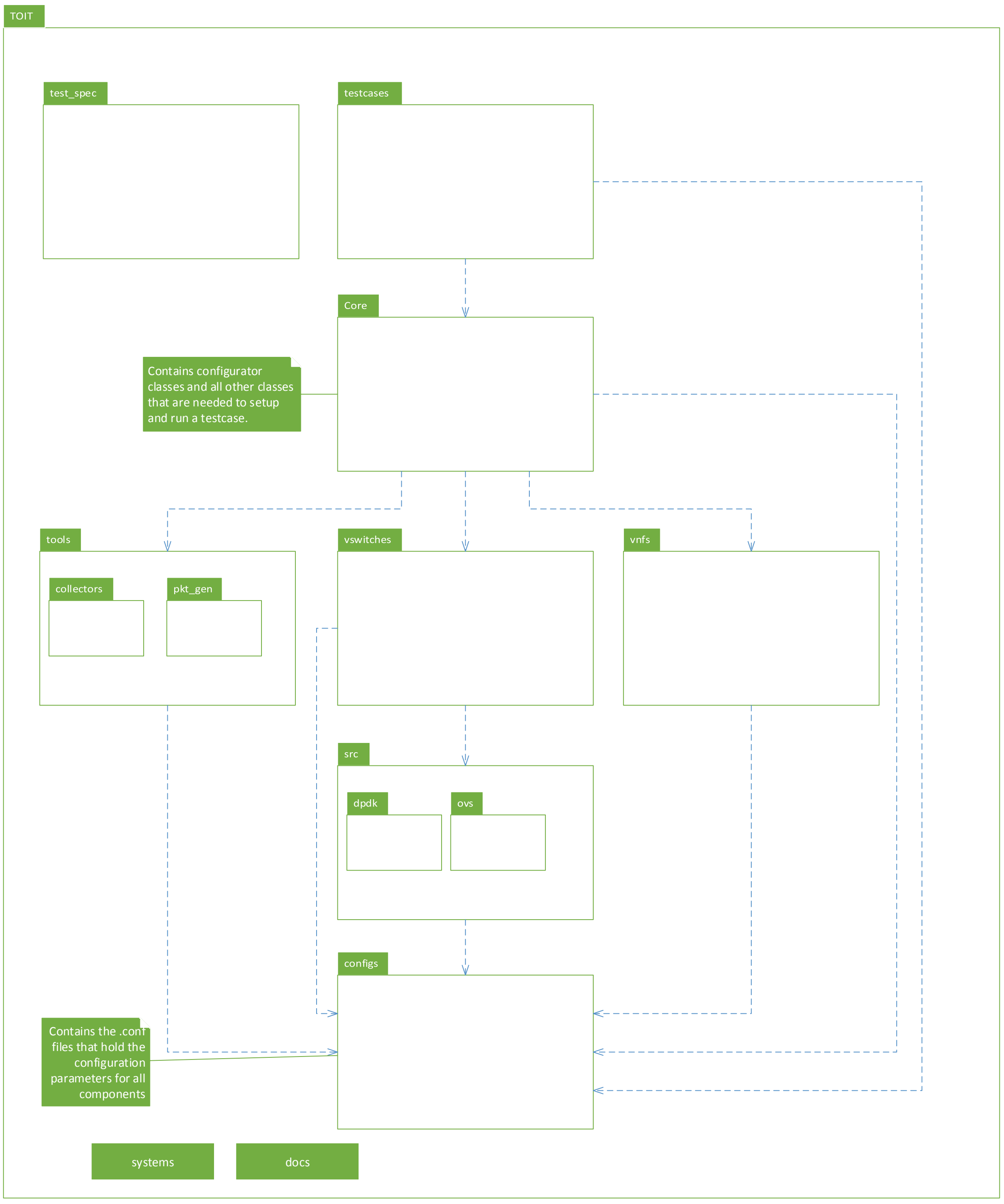

TOIT
test_spec
testcases
Core
Contains configurator classes and all other classes that are needed to setup and run a testcase.
tools
collectors
pkt_gen
vswitches
vnfs
src
dpdk
ovs
configs
Contains the .conf files that hold the configuration parameters for all components
systems
docs

*Figure 6 Development view of TOIT*

## Identify Architectural Patterns

For the TOIT redesign two architectural patterns were chosen, the "Component" architectural pattern and the "Layered" architecture style. The Component architectural pattern supports the identified architectural drivers "Modularity, Modifiability and Extensibility" by design. While the Layered style allows for the division of TOIT into stacked groups, whereby each layer performs a particular job. These layers are: the test specification layer, the test core layer, the base components layer, and the TOIT setup layer. For more information about the advantages and limitations of the chosen patterns, please see Appendix B.

## Chosen Architectural Approaches and decisions

This section will provide an overview of the architectural approaches and decisions. But before diving straight into the approaches and decisions, an understanding of the OO concept of "program to interfaces not implementation" is required. This OO concept works by decoupling a class that uses an interface from the implementation of the classes that realize this interface, thus, allowing a range of implementations to be defined where instances of the interface are used. How this works is that each class that realizes an interface provides some implementation for the method signatures specified in the interface. Any classes using an instance of the interface can instantiate any implementation defined by the classes that realize/implement this interface. As a result the classes that realize this interface can be extended, and any changes are now localized, in other words they do not affect the classes using the interface to access the various implementations. In addition the class using the interface can also be modified without having a ripple effect on the 'implementation classes'.

This concept was particularly applied at the test core layer and the test component layer to exploit the advantages of using this concept which are: increased modifiability and the prevention of the ripple effect through localised change.

**Decisions:**

The new packages that comprise the proposed system are:

- testcases: the purpose of this package is to implement the tests specified in the VSPERF test specification. Each test case will interact with objects in the test core layer.
- Each Test case will be comprised of:
    - Zero or more vSwitch object created by the TOIT.core.ComponentFactory
    - A Traffic object created by the TOIT.core.ComponentFactory
    - Zero or more collectors created by the TOIT.core.ComponentFactory
    - Zero or more VNFs created by the TOIT.core.ComponentFactory

    Traffic results (such as forwarding rate, etc.) for a test case will be returned by the traffic controller object to the test case. The result should include the appropriate column headings for reported results.

Collector results (such as cpu usage, etc.) for a test case will be returned by the collector controller object to the test case. The result should include the appropriate column headings for reported results.

The TestCase object will collect results from the Traffic and Collector objects and write them to file in a defined order so as to be machine readable (Traffic results first followed by Collector results). The Traffic and Collector objects will also write their results independently to stdout but in this case the order will not be defined.

Testcases will retrieve their settings from conf.Testcase.conf.

TestCase pseudo Code:

```
vSwitch_controller(‘‘p2p’’)       # Request a new vSwitch object
                                  # from the component factory

traffic_contoller(‘‘RFC2544’’)    # Request a new traffic object
                                  # from the component factory

traffic.send ()                   # Send traffic


traffic.get_results()             # Get results


write_results_to_file()
```

- core: The purpose of this package is to hide the implementation details of the test components from the test cases and to configure and run a test case. This  configuration includes:
    - Configure the traffic generator.
    - Configure any required loggers.
    - Configure the virtual switch.
    - Configure the VNF if one is required.
    - Send traffic.
    - Report the result in terms of both traffic and any additional loggers/collectors.

  Package abstractions at this layer are not aware of the implementation details of packages at the test components layer. For example core.vSwitch_controller class/object will use a simple API to add a port to a vSwitch add_port(string port), where port can by phy or logical. It’s up to the underlying vSwitch in the vSwitches package to determine what a physical port is (in the case of OVS DPDK this will be a DPDK port).

Packages in core are aware of what a deployment scenario looks like. For example if a test case requests a vSwitch_controller to setup a p2p vSwitch, the vSwitch_ controller knows that it will need to request two phy ports from the vSwitch and install a rule to forward traffic between them.

- tools: will contain general tools used by the tests such as collectors (used collect CPU usage and memory usage information) and traffic generators.
- configs: The purpose of this package is to hold the configuration files (*.conf) that contain the configuration parameters for all the components. This package will expose an API that will allow the retrieval of TOIT settings and test parameters.
- src: the purpose of this package is to configure and build the vSwitch libraries and their dependencies. It's preferred that if a TOIT package or module has a dependency on some additional configuration, it performs this configuration itself. For example test cases that are testing DPDK enabled OVS would require the system to be configured with DPDK prior to setting up a switch. This would include configuring Hugepages and binding interfaces to DPDK. The DPDK vSwitch would be expected to perform this setup in conjunction with the configs package prior to launching the switch.
- vSwitches: the purpose of this package is to configure and set up a virtual switches for a test case.
- VNFs: the purpose of this package is to configure and set up a VNF for a test case.
- docs: the purpose of this package is to hold the documentation that is relevant to the framework.
- test_spec: the purpose of this package is to hold the test specification document.
- systems: this package will look after the installation of the various system packages on which the framework depends such as python …

**Package mapping from Old TOIT to new TOIT and sub package decisions.**

| Old Package/Module | Decision |
|---|---|
| toit.docs | This package will be added to the agreed on directory structure and be updated to reflect the new architecture and design. |
| toit.sysmetrics | This package (as well as its sub packages and modules) will be moved to the tools.collectors package under the new directory structure.<br><br>The modules in this package will implement a new function collect (type), where type can be mem_stats or cpu_stats or other. |
| toit.tests-vSwitchperf | This sub packages and modules in this package will be moved to |

| | |
|---|---|
| | the testcases package in the new directory structure.<br><br>Test cases will be organized based on behaviour. Any test case which relies on the results of a previous test case will test for those results, if the results are not present the test case will print a warning with the prerequisite test to run. |
| toit.trafficgens | This package (as well as its sub-packages and modules) will be moved into tools.pkt_gen in the new directory structure. |
| toit. toit | toit. toit.conf will move to conf in the new directory structure and will contain the configuration files for TOIT. This package will expose an API that will allow the retrieval of TOIT settings and test parameters.<br><br>toit. toit.ovs and its modules will move to vSwitches.ovs_dpdk and will implement a generic vSwitch API that will allow the core.vSwitch_controller object to add a vSwitch, remove a vSwitch, add a port, remove a port, add a flow and remove a flow. At this level (core.vSwitch_conroller) vSwitches, ports and flows are abstracted into simple objects to hide the underlying complexity. For Example a core.vSwitch_controller object only knows about physical or logical ports, it's up to the underlying vSwitch to determine what constitutes a physical port, in the case of OVS_DPDK that is a DPDK type port.<br><br>src.dpdk will also look after configuring DPDK for OVS.<br><br>toit. toit.guest will move to VNFs.qemu<br><br>toit. toit.sysmetrics will be removed as it's now realized as part of tools.collectors.sysmetrics.<br><br>toit. toit.system was configuring and setting up DPDK for OVS. toit. toit.system.system will be moved under src.dpdk and renamed to dpdk.py.<br><br>toit. toit.test.loader will be moved to the core package.<br><br>toit. toit.trafficgen will be moved into tools.pkt_gen package.<br><br>toit. toit.utils.loader please see toit.core.loader row.<br><br>toit. toit.utils.tasks will be moved to core.tasks.<br><br>toit. toit.utils.tests please see toit.core.tests row. |

| | |
|---|---|
| toit. toit.trafficgen.loader<br><br>toit. toit.sysmetrics.loader<br><br>toit. toit.utils.loader | Previously there were separate directories containing loader modules under toit. toit for: traffic generators (old package toit. toit.trafficgen), sysmetrics (old package toit. toit.sysmetrics), and modules (old package toit. toit.utils). These modules implement the same function on a different directory path and thus will be moved to a loader module under core (core.loaders). The function will take the path as an argument. |
| toit. toit.utils.tests | The old module toit. toit.utils.tests simply contains a function to get test parameters, this module will be moved to conf as this package contains the default test parameters and the functions pertaining to retrieving them. |
| toit. toit.conf.Default_settings.py | Default_settings.py will be broken up into several configuration files for TOIT settings, vSwitches, VNFs and traffic.<br><br>Each configuration file will hold the configuration parameters for a particular module. So that if a module is removed, one can simply remove its configuration file from settings and without affecting other configuration files.<br><br>The settings module will be accessed by nearly every package in the system, in order to avoid the upper modules having pass settings structures to lower level modules. |
| | Jobs package/directory will be renamed to configs, as its naming should reflect its purpose which is to hold the module configurations. This package is a collection of configurations for different modules. |

## 3.5 TOIT Redesign

### 3.5.1 Introduction

### 3.5.2 TOIT use cases

TOIT use cases have not changed as part of the redesign and remain pretty straight forward.

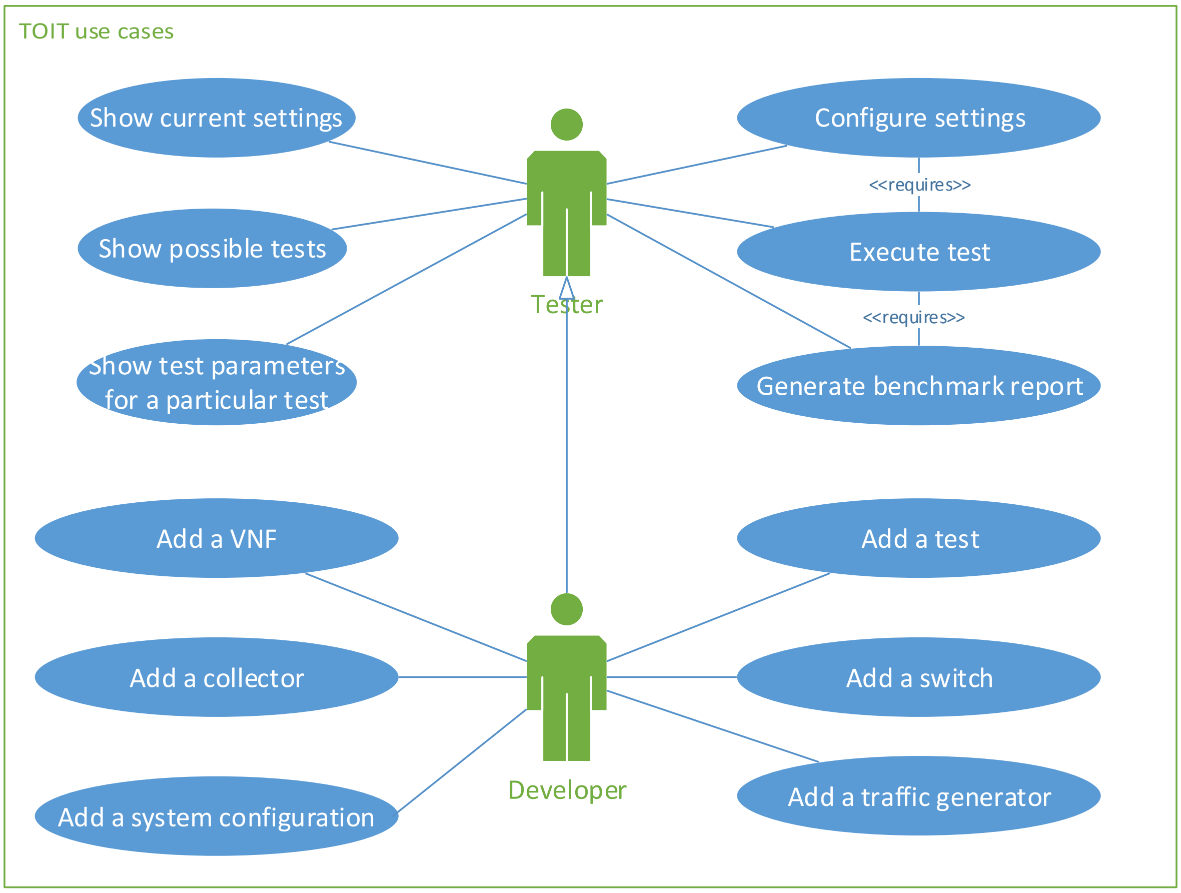


*Figure 7: TOIT use cases*

#### *3.5.2.1 TOIT use case descriptions*

| | |
|---|---|
| Use Case Number | 1 |
| Use Case Name | Show current settings |
| Description | As a Tester I would like to print the current settings in order to see which settings have been, and can be enabled in my system |
| Actors | Tester, Developer |
| Preconditions | A settings hierarchy must exist and there must be a way to print them to screen, or to a file. |
| Post conditions | The settings have been displayed on screen or are available in a |

| | |
|---|---|
| | file. |
| Flow of activities | 1. Tester runs a command to output the current settings profile<br>2. Tester modifies the system through the configuration files.<br>3. Tester re-outputs the settings to see how they have been changed. |
| Exception condition | |

| | |
|---|---|
| Use Case Number | 2 |
| Use Case Name | Show possible tests |
| Description | As a Tester I would like to be able to print all possible tests to screen. |
| Actors | Tester, Developer |
| Preconditions | There exists a way to list all tests and their descriptions. |
| Post conditions | Tests and their descriptions have been output to screen. |
| Flow of activities | 1. Tester runs a command to list tests.<br>2. Tester can then choose from the list of tests which one they would like to run. |
| Exception condition | No tests exist. |

| | |
|---|---|
| Use Case Number | 3 |
| Use Case Name | Show test parameters for a particular test |
| Description | As a Tester I would like to show the potential parameters for a test, so that I could properly configure it. |
| Actors | Tester, Developer |
| Preconditions | Test parameters exist for a particular test.<br>The Tester knows the name of the test they wish to call. |
| Post conditions | The test parameters for the listed test have been output to screen. |
| Flow of activities | 1. The Tester selects their required test.<br>2. The Tester runs a command to print the test parameters for the selected test.<br>3. The Tester can select which of the parameters they wish to override and can pass them when running the test via --test-params. |
| Exception condition | No test parameters exist. Return this. |

| | |
|---|---|
| Use Case Number | 4 |
| Use Case Name | Configure settings |

| | |
|---|---|
| Description | As a tester I would like to modify:<br>- The parameters and settings used for running the tests.<br>- The settings of low level components. |
| Actors | Tester, Developer |
| Preconditions | Configuration files exist |
| Post conditions | New settings are defined in a configuration file. |
| Flow of activities | A developer or tester creates or modifies the configuration files for the relevant subcomponent. |
| Exception condition | |

| | |
|---|---|
| Use Case Number | 5 |
| Use Case Name | Execute test |
| Description | As a tester I would like to be able to run a specific test or tests and collect the test results. |
| Actors | Tester, Developer |
| Preconditions | The test the developer or tester wishes to run is implemented |
| Post conditions | The results are available to the tester/developer |
| Flow of activities | 1. The tester or developer runs a command to see a list of what tests are available and identifies the test(s) they want to run.<br>2. The tester or developer runs the test by test name or category. |
| Exception condition | The desired test does not exist. The developer would need to implement the test through a new patch submitted to the framework. |

| | |
|---|---|
| Use Case Number | 6 |
| Use Case Name | Generate benchmark report. |
| Description | As a tester I would like to generate a benchmark report from the results of a test that was run. |
| Actors | Tester, Developer |
| Preconditions | A test was run. |
| Post conditions | A benchmark report is available. |
| Flow of activities | 1. The tester or developer runs the test by test name or category.<br>2. The tester or developer generates the report with the test results. |
| Exception condition | The desired test does not exist. The developer would need to |

| | |
|---|---|
| | implement the test through a new patch submitted to the framework. The patch should also update the results reporting mechanism so results can be collected. |

| | |
|---|---|
| Use Case Number | 7 |
| Use Case Name | Add a test |
| Description | As a developer I would like to be able to add support for a new test so that a tester can run this new test without affecting the operation of any existing virtual switches. |
| Actors | Developer |
| Preconditions | Modular implementation of tests. |
| Post conditions | New test definition which allows the use of a new type of test. |
| Flow of activities | 1. Developer creates new testcase by modifying the testcases.conf file to include their new test.<br>2. Developer implements any additional behaviours/infrastructure required for the test case in the framework.<br>3. Developer modifies and additional configurations that are required to run the test.<br>4. Developer updates the existing framework documentation. |
| Exception condition | The Developer wishes to add a test operation to the framework which does not already exis. In this case, they will first submit a separate patch to the repo to enable this feature, if possible, for all existing tests, before implementing their new test in a separate patch. |

| | |
|---|---|
| Use Case Number | 8 |
| Use Case Name | Add a vSwitch |
| Description | As a developer I would like to be able to add support for a virtual switch so that a tester can test a new type of virtual switch without affecting the operation of any existing virtual switches. |
| Actors | Developer |
| Preconditions | Modular implementation of virtual switches. Single API that all virtual switches conform to. |
| Post conditions | New virtual switch definition which allows the use of a new type of virtual switch. |
| Flow of activities | 1. Developer creates new folder within vSwitches/<br>2. Developer creates a file in that folder.<br>3. Developer implements the functions declared in the vSwitch interface class.<br>4. Developer adds the necessary configuration to the default configuration file. |

| | |
|---|---|
| | 5. Developer updates the existing framework documentation. |
| Exception condition | The Developer wishes to add a virtual switch operation to the framework which does not already exist as an abstract method. In this case, they will first submit a separate patch to the repo to enable this feature, if possible, for all existing virtual switches, before implementing their new virtual switch in a separate patch. |

| | |
|---|---|
| Use Case Number | 9 |
| Use Case Name | Add a traffic generator |
| Description | As a developer I would like to be able to add support for a traffic generator so that a tester can send and receive traffic using a new type of traffic generator without affecting the operation of any existing traffic generators. |
| Actors | Developer |
| Preconditions | Modular implementation of traffic generators. Single API that all traffic generators conform to. |
| Post conditions | A new implementation of a traffic generator which enables the tester to send and receive traffic from a new type of traffic generator. |
| Flow of activities | 1. Developer creates new package within tools/trafficgens/<br>2. Developer implements abstract methods defined by the Trafficgen interface class in a new python module.<br>3. Developer adds the necessary configuration to the default configuration file.<br>4. Developer updates the existing framework documentation. |
| Exception condition | The Developer wishes to add a Trafficgen operation to the framework which does not already exist as an abstract method. In this case, they will first submit a separate patch to the repo to enable this feature, if possible, for all existing traffic generators, before implementing their new traffic generator in a separate patch. |

| | |
|---|---|
| Use Case Number | 10 |
| Use Case Name | Add a VNF/hypervisor |
| Description | As a developer, I would like be able to add support for a hypervisor/VNF so that a tester can use VNFs, without affecting any existing implementations of other hypervisors/VNFS. |
| Actors | Developer |
| Preconditions | Modular implementation of hypervisors/VNFs. Single API that all hypervisors/VNFs conform to. |
| Post conditions | New VNF/hypervisor definition which allows the use of a new |

| | |
|---|---|
| | type of, or version of, hypervisor. |
| Flow of activities | 1. Developer creates new package within vnfs/<br>2. Developer implements abstract methods defined by the VNF interface class in a new python module.<br>3. Developer adds the necessary configuration to the default configuration file.<br>4. Developer updates the existing framework documentation. |
| Exception condition | The Developer wishes to add a VNF operation to the framework which does not already exist as an abstract method. In this case, they will first submit a separate patch to the repo to enable this feature, if possible, for all existing hypervisors, before implementing their new VNF in a separate patch. |

| | |
|---|---|
| Use Case Number | 11 |
| Use Case Name | Add a collector |
| Description | As a developer, I would like to be able to add support for a collector so that a tester can collect different system metrics. This should be done without affecting the operation of any existing collectors. |
| Actors | Developer |
| Preconditions | Modular definition of collectors. Single API that all collectors conform to. |
| Post conditions | New collector implementation which allows the use of a new type of system metrics logger. |
| Flow of activities | 1. Developer creates new folder within tools/collectors/<br>2. Developer implements abstract methods defined by the Collector interface class.<br>3. Developer adds the necessary configuration to the default configuration file.<br>4. Developer updates the existing framework documentation. |
| Exception condition | The Developer wishes to add a collector operation to the framework which does not already exist as an abstract method. In this case, they will first submit a separate patch to the repo to enable this feature, if possible, for all existing collectors, before implementing their new collector in a separate patch. |

| | |
|---|---|
| Use Case Number | 12 |
| Use Case Name | Add a system configuration |
| Description | As a developer I would like to enable the framework on a new platform without affecting the operation of the framework on existing platforms. |
| Actors | Developer |

| | |
|---|---|
| Preconditions | Modular definition of a system configuration provider. Single API that all system configuration providers conform to. |
| Post conditions | New system configuration provider which enables the framework to be run on a new platform. |
| Flow of activities | 1. Developer creates new conf file within the conf/ directory<br>2. Developer implements the appropriate methods to retrieve the configuration.<br>3. Developer updates the existing framework documentation. |
| Exception condition | The Developer wishes to add a new configuration option the framework which does not already exist. In this case, they will implement their new system configuration and submit in patch. |

### 3.5.3 Interface Identification

According to Cheesman et el. [7] the identification of system interfaces is achieved through the decomposition of use cases into steps in order to identify the operations/responsibilities the system needs to fulfil. The primary system interfaces are represented by Figure 9. These classes represent the interfaces of the test core components. They represent a higher level interface that abstracts the complex interfaces of the low level components (vSwitches, VNFs etc.) from the TestCase.

As for the test component interfaces, they are identified by refining the steps in the decomposed use-case in order to identify the information and operations the core and components need to manage.

#### *3.5.3.1 Package decomposition:*

##### Package: testcases

Contains the class that will be created in the main loop that loops over tests in the testcase.conf file and then proceeds to create the TestCase object and run the test.

Pseudo code:

```
Main() {
   Foreach testconf in testcase.conf :
      Testcase tc = new Testcase();
      tc.run(testconf, trafficconf, vnfconf, customconf);
}
```

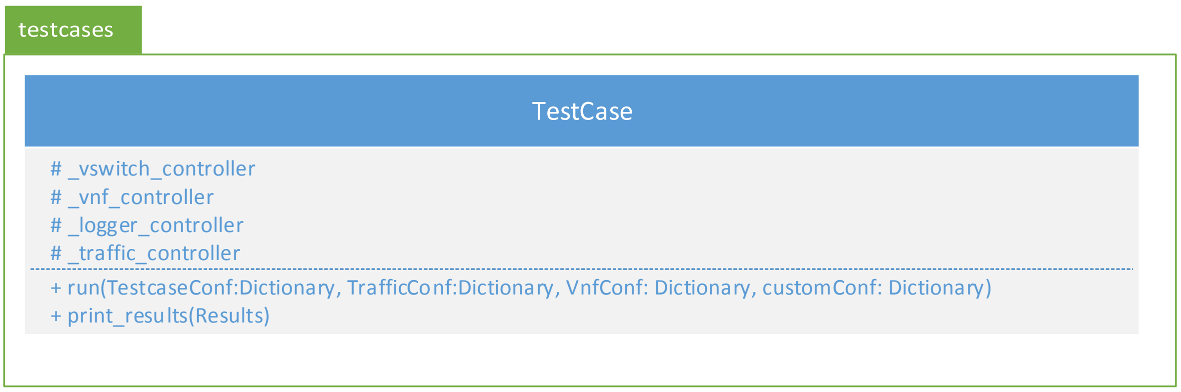


*Figure 8 TestCase class diagram*

TestCaseConf is a dictionary contained in testcases.conf and looks like:

```
Performance_tests = {
        "<Test_Case_ID>": {
          "Type": TestTypes.[THROUGHPUT|BACK2BACK|STRESS]>,
          "Traffic Type": [cont|burst|rfc2544|back2back],
           “MultiStream”: [true|false],
          “Test modifier”: [FrameMod|Other],
          "Logger": [cpu|memory|None],
          "Dependency": [Test_Case_ID |None],
          "Deployment": [p2p|pvp|pvvp|pv|vp|vpv| a comma separated list of
        deployments that apply],
       },
}
```

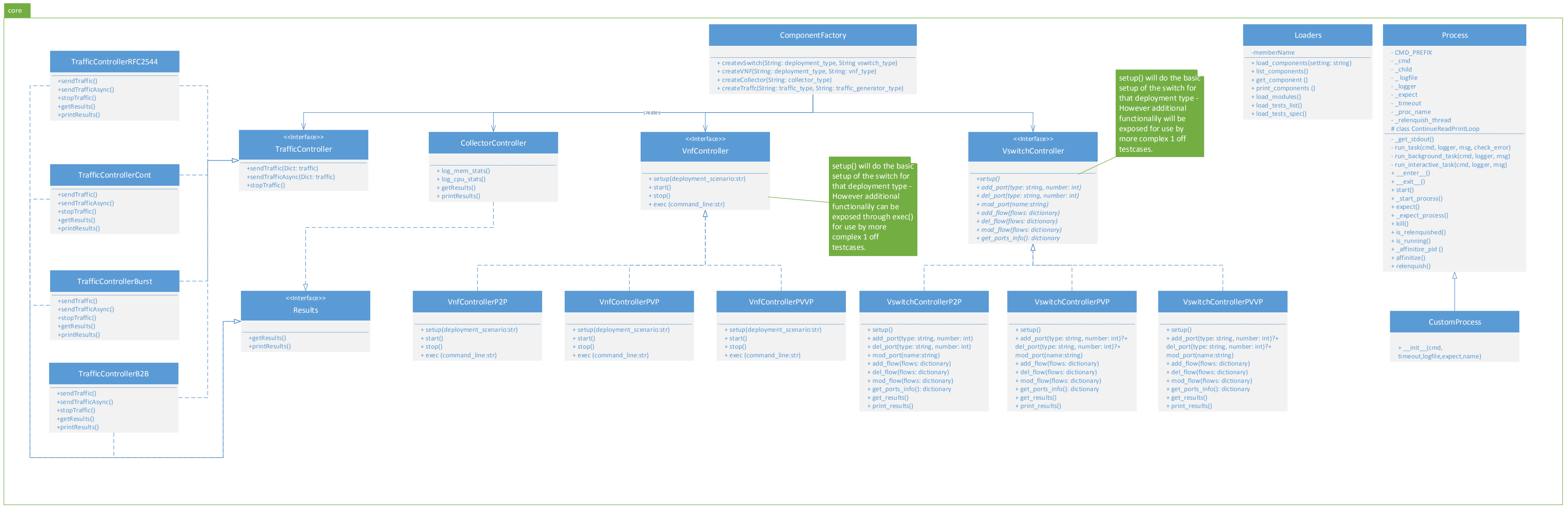
The other conf parameters will be discussed under the Pacakge: conf section.

Each TestCase will configure a vSwitch, a traffic Item, a VNF and a logger through a set of controllers that reside in the core package.

## Package: core

Controllers will be used to configure and run the various sub-components that a TestCase requires. These controllers abstract the underlying implementation details of the sub-components, such as

Switches, traffic, VNFs, collectors, and expose a simplified API to the TestCase. Controllers are created through the Component Factory.



core
ComponentFactory
+ createvSwitch(String: deployment_type, String vswitch_type)
+ createVNF(String: deployment_type, String: vnf_type)
+ createCollector(String: collector_type)
+ createTraffc(String: traffic_type, String: traffic_generator_type)
creates
TrafficControllerRFC2544
+sendTraffic()
+sendTrafficAsync()
+stopTraffic()
+getResults()
+printResults()
TrafficControllerCont
+sendTraffic()
+sendTrafficAsync()
+stopTraffic()
+getResults()
+printResults()
TrafficControllerBurst
+sendTraffic()
+sendTrafficAsync()
+stopTraffic()
+getResults()
+printResults()
TrafficControllerB2B
+sendTraffic()
+sendTrafficAsync()
+stopTraffic()
+getResults()
+printResults()
<<Interface>>
TrafficController
+sendTraffic(Dict: traffic)
+sendTrafficAsync(Dict: traffic)
+stopTraffic()
CollectorController
+ log_mem_stats()
+ log_cpu_stats()
+ getResults()
+ printResults()
<<Interface>>
Results
+getResults()
+printResults()
<<Interface>>
VnfController
+ setup(deployment_scenario:str)
+ start()
+ stop()
+ exec (command_line:str)
setup() will do the basic setup of the switch for that deployment type - However additional functionally can be exposed through exec() for use by more complex 1 off testcases.
VnfControllerP2P
+ setup(deployment_scenario:str)
+ start()
+ stop()
+ exec (command_line:str)
VnfControllerPVP
+ setup(deployment_scenario:str)
+ start()
+ stop()
+ exec (command_line:str)
VnfControllerPVVP
+ setup(deployment_scenario:str)
+ start()
+ stop()
+ exec (command_line:str)
<<Interface>>
VswitchController
+setup()
+ add_port(type: string, number: int)
+ del_port(type: string, number: int)
+ mod_port(name:string)
+ add_flow(flows: dictionary)
+ del_flow(flows: dictionary)
+ mod_flow(flows: dictionary)
+ get_ports_info(): dictionary
setup() will do the basic setup of the switch for that deployment type - However additional functionally will be exposed for use by more complex 1 off testcases.
VswitchControllerP2P
+ setup()
+ add_port(type: string, number: int)
+ del_port(type: string, number: int)
+ mod_port(name:string)
+ add_flow(flows: dictionary)
+ del_flow(flows: dictionary)
+ mod_flow(flows: dictionary)
+ get_ports_info(): dictionary
+ get_results()
+ print_results()
VswitchControllerPVP
+ setup()
+ add_port(type: string, number: int)?+
del_port(type: string, number: int)?+
mod_port(name:string)
+ add_flow(flows: dictionary)
+ del_flow(flows: dictionary)
+ mod_flow(flows: dictionary)
+ get_ports_info(): dictionary
+ get_results()
+ print_results()
VswitchControllerPVVP
+ setup()
+ add_port(type: string, number: int)?+
del_port(type: string, number: int)?+
mod_port(name:string)
+ add_flow(flows: dictionary)
+ del_flow(flows: dictionary)
+ mod_flow(flows: dictionary)
+ get_ports_info(): dictionary
+ get_results()
+ print_results()
Loaders
-memberName
+ load_components(setting: string)
+ list_components()
+ get_component ()
+ print_components ()
+ load_modules()
+ load_tests_list()
+ load_tests_spec()
Process
- CMD_PREFIX
- _cmd
- _child
- _logfile
- _logger
- _expect
- _timeout
- _proc_name
- _relenquish_thread
# class ContinueReadPrintLoop
- _get_stdout()
- run_task(cmd, logger, msg, check_error)
- run_background_task(cmd, logger, msg)
- run_interactive_task(cmd, logger, msg)
+ __enter__()
+ __exit__()
+ start()
+ _start_process()
+ expect()
+ _expect_process()
+ kill()
+ is_relenquished()
+ is_running()
+ _affinitize_pid ()
+ affinitize()
+ relenquish()
CustomProcess
+ __init__(cmd, timeout,logfile,expect,name)

*Figure 9 Core Package Class diagram*

## Package: conf

The conf package will contain the configuration files for all system components, it will also contain a Configuration class that will retrieve component settings from the configuration files. It will breakdown the settings.py into its constituent parts: testcases.conf, vswitches.conf, vnf.conf, traffic.conf as well as a custom.conf for any additional configurations that lie outside the remit of the other configuration files. These configurations will be read by the main loop and passed to the TestCase based on the TestCase configuration, i.e. if a TestCase does not require a VNF, the VnfConf is not read and passed as "NONE" to setup_test().

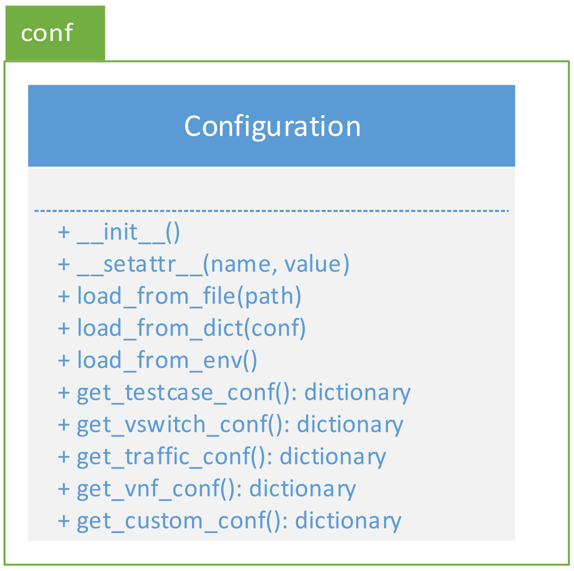

conf
Configuration
+ __init__()
+ __setattr__(name, value)
+ load_from_file(path)
+ load_from_dict(conf)
+ load_from_env()
+ get_testcase_conf(): dictionary
+ get_vswitch_conf(): dictionary
+ get_traffic_conf(): dictionary
+ get_vnf_conf(): dictionary
+ get_custom_conf(): dictionary

*Figure 10 Configuration Class Diagram*

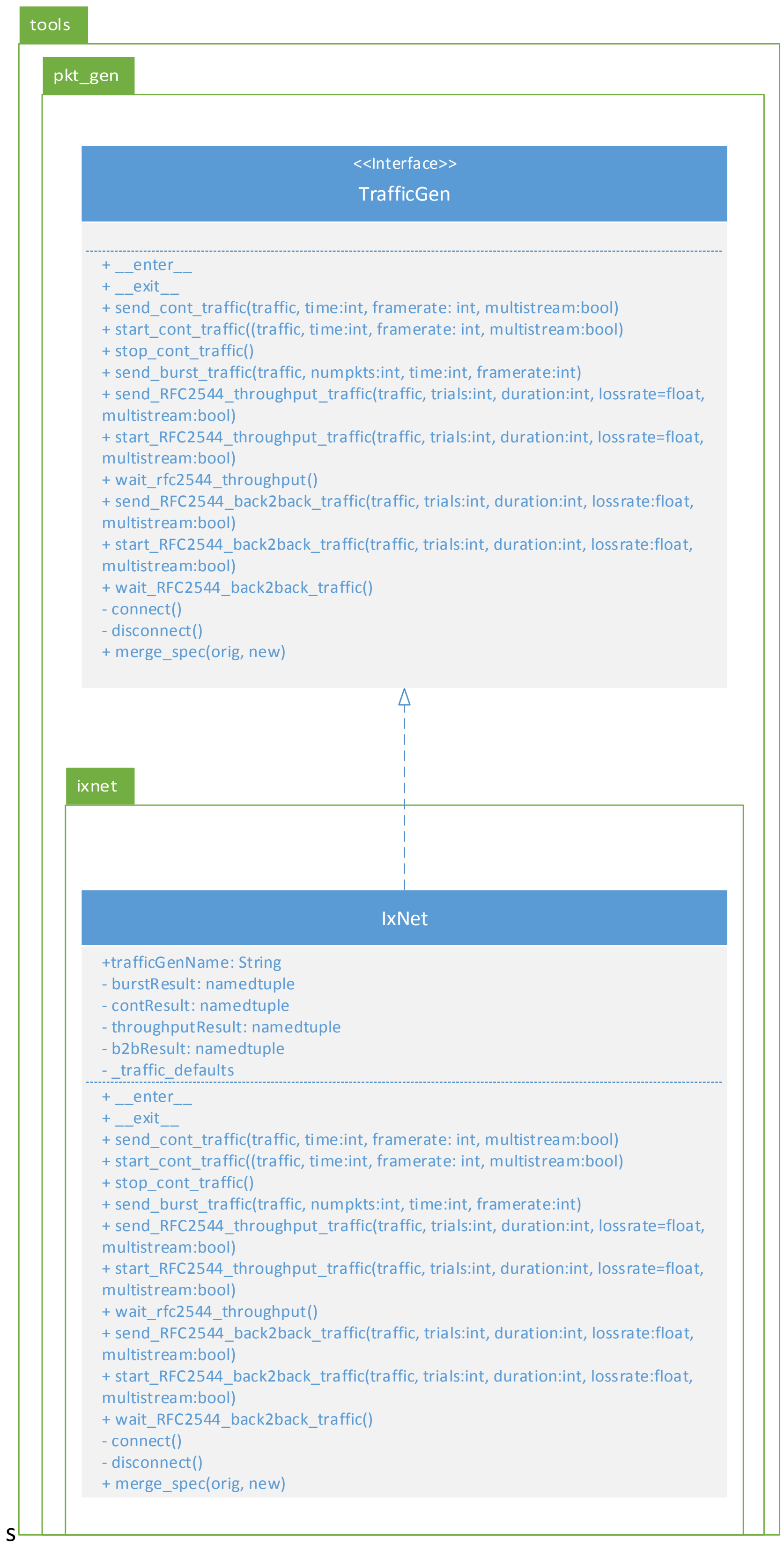
Each sub-component package will contain an __init__.py file that will select the default component from the components configuration file (or from a command line parameter). For example vswitches/__init__.py:

```python
from vswitchperf import conf

DEFAULT_VSWITCH = 'ovs_dpdk_vhost'


def get_vswitch():
    """
    Load the vswitch specified as a test parameter.

    This interface allows the user to determine which vSwitch to use by
    way of a test param, e.g.:

        ./run --test-param vswitch=ovs_dpdk_vhost

    If this is not specified, the default vSwitch implementation will
    be loaded. If an invalid vSwitch is specified an exception will be
    raised.
    """
    vswitch_mod_path = '.'.join(
        ['_vswitches', utils.get_test_param('vswitch', DEFAULT_VSWITCH)])

    return __import__(vswitch_mod_path, fromlist=[''])

vswitch = get_vswitch()
```

Loaders will retrieve the class of choice (for a subcomponent) through these functions.

### Package: tools.pkt_gen.ixnet

Exposes a simple traffic generator interface that can be configured by the traffic controller.



tools
pkt_gen
<<Interface>>
TrafficGen
+ __enter__
+ __exit__
+ send_cont_traffic(traffic, time:int, framerate: int, multistream:bool)
+ start_cont_traffic((traffic, time:int, framerate: int, multistream:bool)
+ stop_cont_traffic()
+ send_burst_traffic(traffic, numpkts:int, time:int, framerate:int)
+ send_RFC2544_throughput_traffic(traffic, trials:int, duration:int, lossrate=float, multistream:bool)
+ start_RFC2544_throughput_traffic(traffic, trials:int, duration:int, lossrate=float, multistream:bool)
+ wait_rfc2544_throughput()
+ send_RFC2544_back2back_traffic(traffic, trials:int, duration:int, lossrate:float, multistream:bool)
+ start_RFC2544_back2back_traffic(traffic, trials:int, duration:int, lossrate:float, multistream:bool)
+ wait_RFC2544_back2back_traffic()
- connect()
- disconnect()
+ merge_spec(orig, new)
ixnet
IxNet
+trafficGenName: String
- burstResult: namedtuple
- contResult: namedtuple
- throughputResult: namedtuple
- b2bResult: namedtuple
- _traffic_defaults
+ __enter__
+ __exit__
+ send_cont_traffic(traffic, time:int, framerate: int, multistream:bool)
+ start_cont_traffic((traffic, time:int, framerate: int, multistream:bool)
+ stop_cont_traffic()
+ send_burst_traffic(traffic, numpkts:int, time:int, framerate:int)
+ send_RFC2544_throughput_traffic(traffic, trials:int, duration:int, lossrate=float, multistream:bool)
+ start_RFC2544_throughput_traffic(traffic, trials:int, duration:int, lossrate=float, multistream:bool)
+ wait_rfc2544_throughput()
+ send_RFC2544_back2back_traffic(traffic, trials:int, duration:int, lossrate:float, multistream:bool)
+ start_RFC2544_back2back_traffic(traffic, trials:int, duration:int, lossrate:float, multistream:bool)
+ wait_RFC2544_back2back_traffic()
- connect()
- disconnect()
+ merge_spec(orig, new)

s

*Figure 11 TrafficGen Class Diagram*

### Package: tools.collectors

Exposes a simple collector that can be configured by the collector controller.

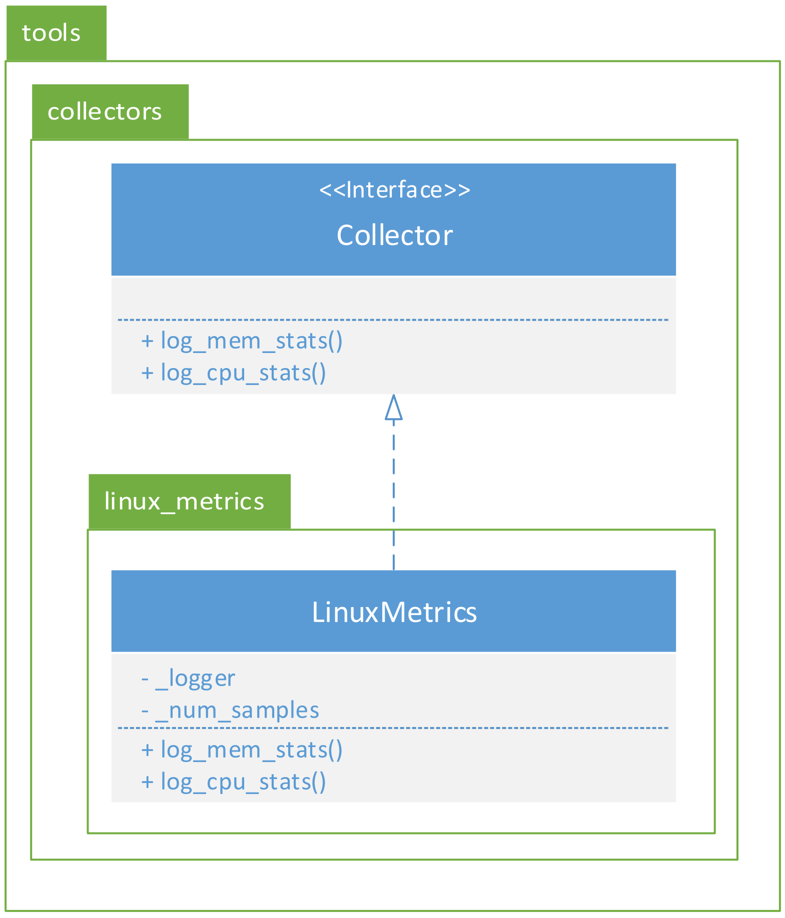
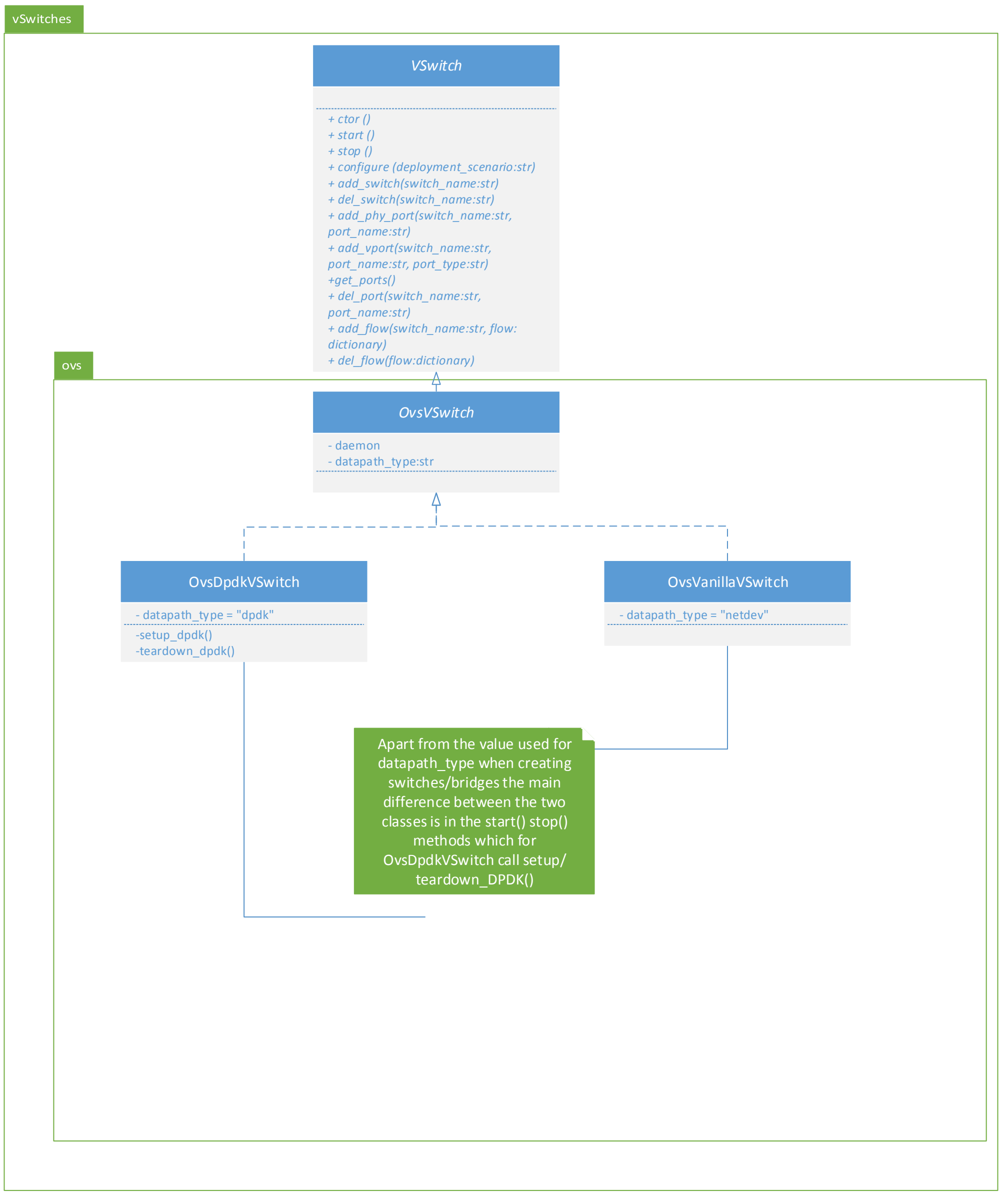


*Figure 12 Collectors Class Diagram*

## Package: vSwitches

Exposes a simple vSwitch that can be configured by the vSwitch controller.


vSwitches
VSwitch
+ ctor ()
+ start ()
+ stop ()
+ configure (deployment_scenario:str)
+ add_switch(switch_name:str)
+ del_switch(switch_name:str)
+ add_phy_port(switch_name:str, port_name:str)
+ add_vport(switch_name:str, port_name:str, port_type:str)
+get_ports()
+ del_port(switch_name:str, port_name:str)
+ add_flow(switch_name:str, flow: dictionary)
+ del_flow(flow:dictionary)
ovs
OvsVSwitch
- daemon
- datapath_type:str
OvsDpdkVSwitch
- datapath_type = "dpdk"
-setup_dpdk()
-teardown_dpdk()
OvsVanillaVSwitch
- datapath_type = "netdev"

Apart from the value used for datapath_type when creating switches/bridges the main difference between the two classes is in the start() stop() methods which for OvsDpdkVSwitch call setup/ teardown_DPDK()

*Figure 13 vSwitches Class Diagram*

## Package: VNFs

Exposes an interface to VNF that can be configured by the VNF controller.

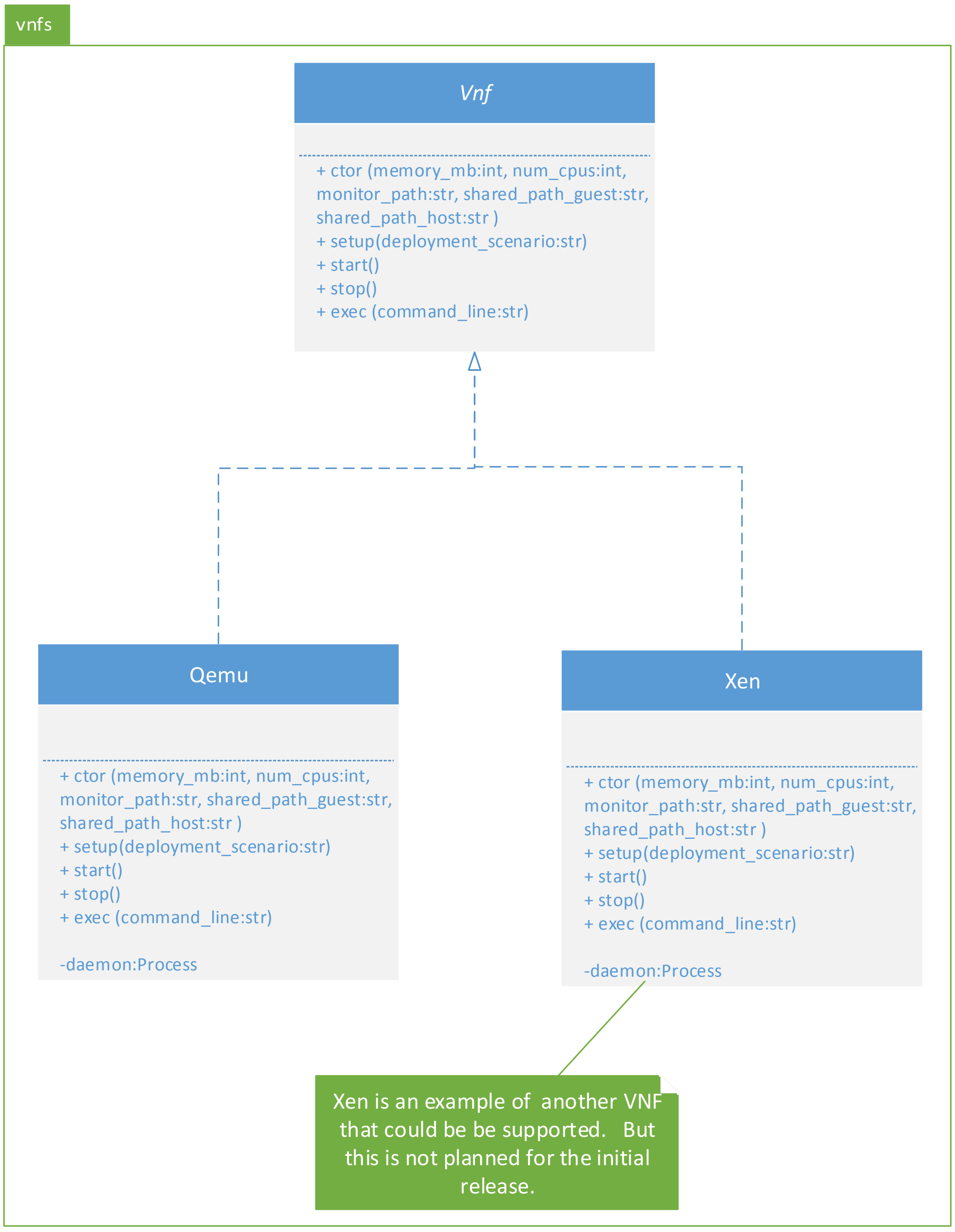

vnfs
Vnf
+ ctor (memory_mb:int, num_cpus:int, monitor_path:str, shared_path_guest:str, shared_path_host:str )
+ setup(deployment_scenario:str)
+ start()
+ stop()
+ exec (command_line:str)
Qemu
+ ctor (memory_mb:int, num_cpus:int, monitor_path:str, shared_path_guest:str, shared_path_host:str )
+ setup(deployment_scenario:str)
+ start()
+ stop()
+ exec (command_line:str)
-daemon:Process
Xen
+ ctor (memory_mb:int, num_cpus:int, monitor_path:str, shared_path_guest:str, shared_path_host:str )
+ setup(deployment_scenario:str)
+ start()
+ stop()
+ exec (command_line:str)
-daemon:Process
Xen is an example of another VNF that could be be supported. But this is not planned for the initial release.

*Figure 14 VNFs class diagram*

Please note that the Qemu class here will be ported from existing TOIT as is and not all its functions are presented in the class diagram.

## Package: src.dpdk

Exposes functions that allow components to configure DPDK on the host. It also exposes a function to copy DPDK from the host to the guest.

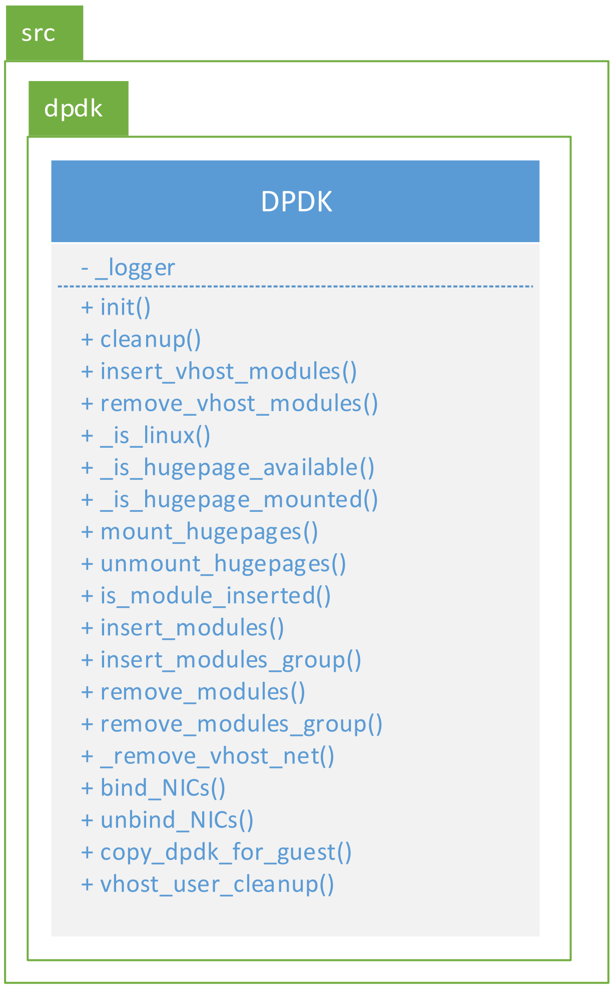


*Figure 15 DPDK Class Diagram*

## Package: src.ovs

This package will exposes functions that allow components to configure OVS on the host, and allow users to use OVS tools such as ovs-vsctl or ovs-ofctl. The classes will re-use as much as possible of the vswitchd class and the ofctl class in existing TOIT.

## P2P setup sequence diagram

The following sequence diagrams presents the general program flow to setup a TestCase.

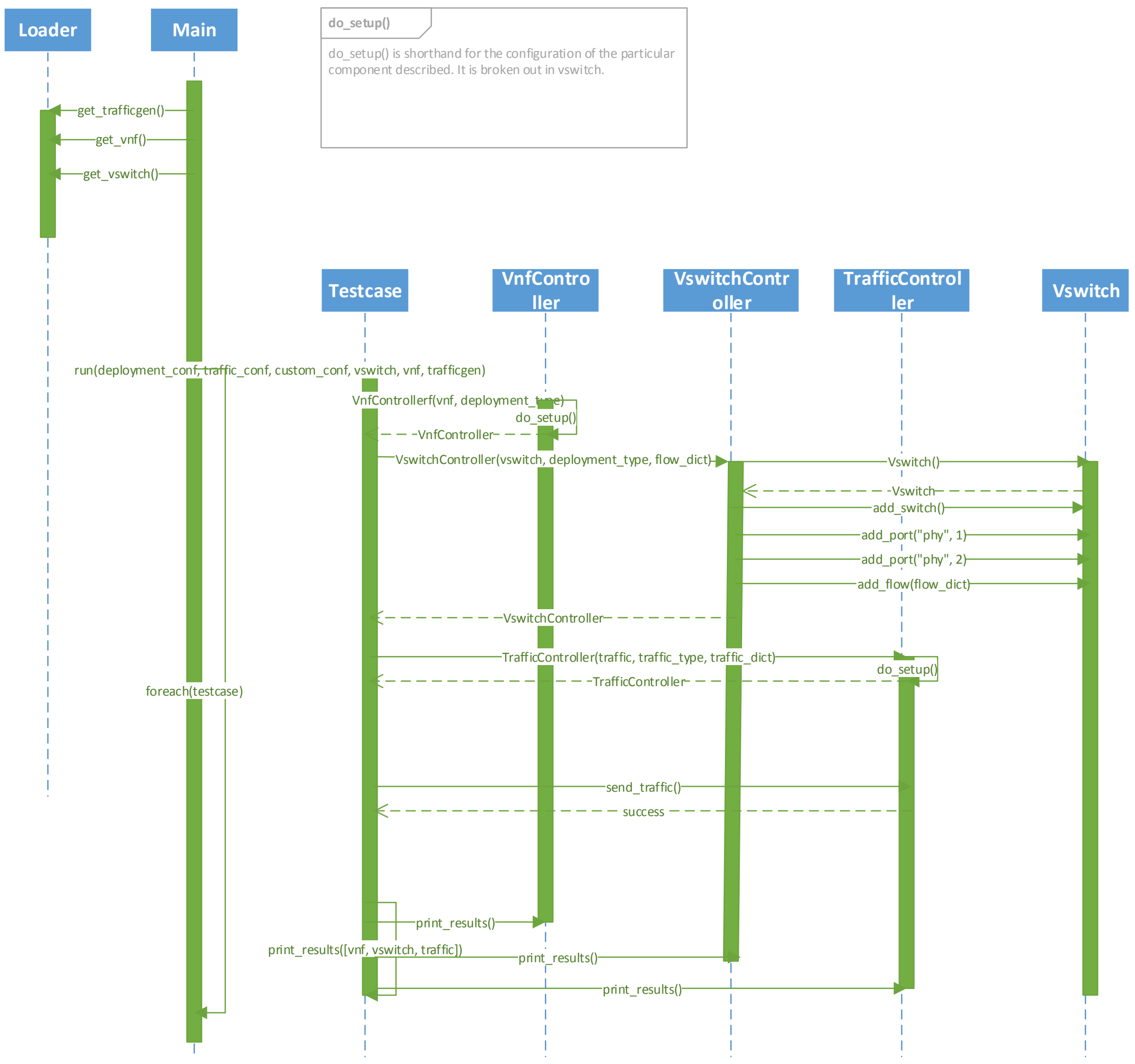

Loader
Main
do_setup()
do_setup() is shorthand for the configuration of the particular component described. It is broken out in vswitch.
get_trafficgen()
get_vnf()
get_vswitch()
Testcase
VnfController
VswitchController
TrafficController
Vswitch
run(deployment_conf, traffic_conf, custom_conf, vswitch, vnf, trafficgen)
VnfControllerf(vnf, deployment_type)
do_setup()
VnfController
VswitchController(vswitch, deployment_type, flow_dict)
Vswitch()
Vswitch
add_switch()
add_port("phy", 1)
add_port("phy", 2)
add_flow(flow_dict)
VswitchController
TrafficController(traffic, traffic_type, traffic_dict)
do_setup()
TrafficController
foreach(testcase)
send_traffic()
success
print_results()
print_results([vnf, vswitch, traffic])
print_results()
print_results()

*Figure 16: Sequence Diagram of a Test Run*

### Directory Tree with new packages

```
.
├── configs
├── core
├── docs
├── src
|    ├── dpdk
|    ├── mk
|    └── ovs
├── systems
|    ├── CentOS
|    └── Ubuntu
├── testcases
├── test_spec
├── tools
|    ├── collectors
|           └── linux-metrics
|    └── pkt_gen
|            └── ixnet
├── vnfs
|    └── qemu
└── vswitches
|    └── ovs
```

## Open Issues

# *Appendix A: Architecture Trade-off Analysis Method (ATAM)*

Architecture Trade-off Analysis Method (ATAM) is a method for the identification and evaluation of a software architecture. This method provides scope for the identification, documentation and evaluation of the architecture of TOIT, particularly how well the suggested architecture meets TOIT's quality attributes [3], as well as the identification of any possible risks early in the lifecycle. Generally it involves an evaluation team, architects and project stakeholders [4]. For the application of this method, the authors, as well as the VSPERF committers and contributors will assume these roles as needs be.

ATAM can be broken in to four distinct phases, Figure 7 shows the conceptual flow of the ATAM method.

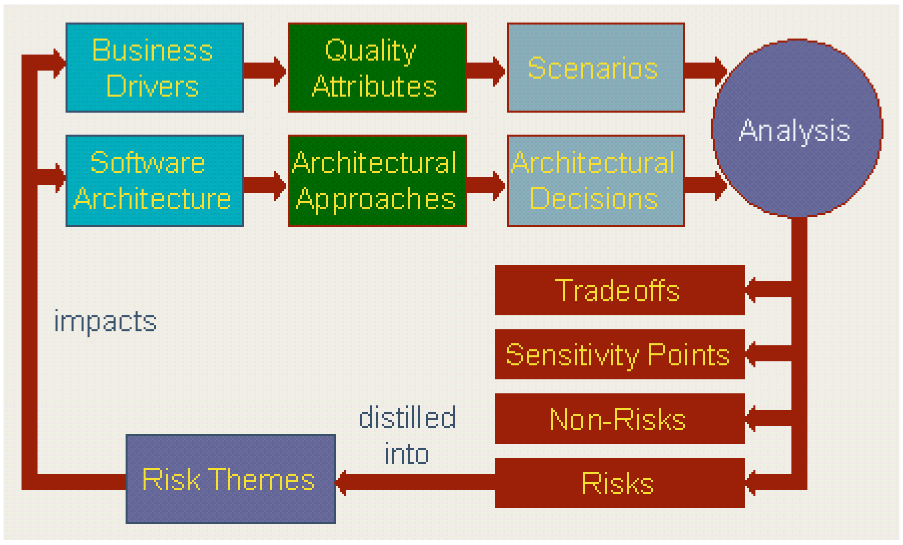


*Figure 17 Conceptual flow of ATAM [4]*

A summary of steps that will be applied in each of the phases as specified by Clements et al. [3]:

**Steps involved in presentation:**

- Presenting the ATAM.

- Presenting the system’s business drivers: The business drivers help identify the system’s quality attributes, which can be converted into specific use cases.
- Presenting the system’s architecture: The presentation of the software architecture shows the various architectural decisions and approaches undertaken to meet the identified quality attributes.

**Steps involved in investigation and analysis:**

- Identifying architectural patterns and decisions including: system sensitivity points, trade-offs, risks and non-risks are identified during this phase
- Assessing the chosen architectural approach in terms of achieving the system’s quality attributes
- Generating the Utility Tree.

**Steps involved in testing**:

- Checking the results of the Investigation and Analysis phase against the stakeholders’ needs. This will be achieved through the review of this document.

**Steps involved in Reporting**:

- Presenting the results of the evaluation. This will be achieved through the presentation of the rework of this document.

# Appendix B: Advantages and limitations of the selected patterns

The core principals and advantages of the Component architectural pattern are the following [0]:

- Reusable: Components are designed to be reused in different situations. Not all components, however, have to be completely reusable and may be designed with specific tasks in mind.
- Replaceable: Components which implement the same interfaces can be used interchangeably. This is key for the packages in the base components layer. For example vSwitch implementations need to be replaceable/pluggable under the hood of the vswitches package.
- Not context specific: Information on the context (state) of the system should be passed to the components, where they rely on it, so as to avoid dependency on a particular configuration, running order or setup. This also makes it easy to write tests for individual components.
- Extensible: One should be able to extend the behaviour of a component using existing components.
- Encapsulated: Interactions with Components are defined only by their interfaces, as such, they do not reveal any of their “inner workings” or variables. This is of particular interest at the border of the test core and the test components layers. For example a vSwitch abstraction in the test core layer should be able to configure a vSwitch without worrying about the underlying vSwitch implementation.
- Independent: Dependencies are minimised between components, as such they can be used in new circumstances without affecting any of the existing components.

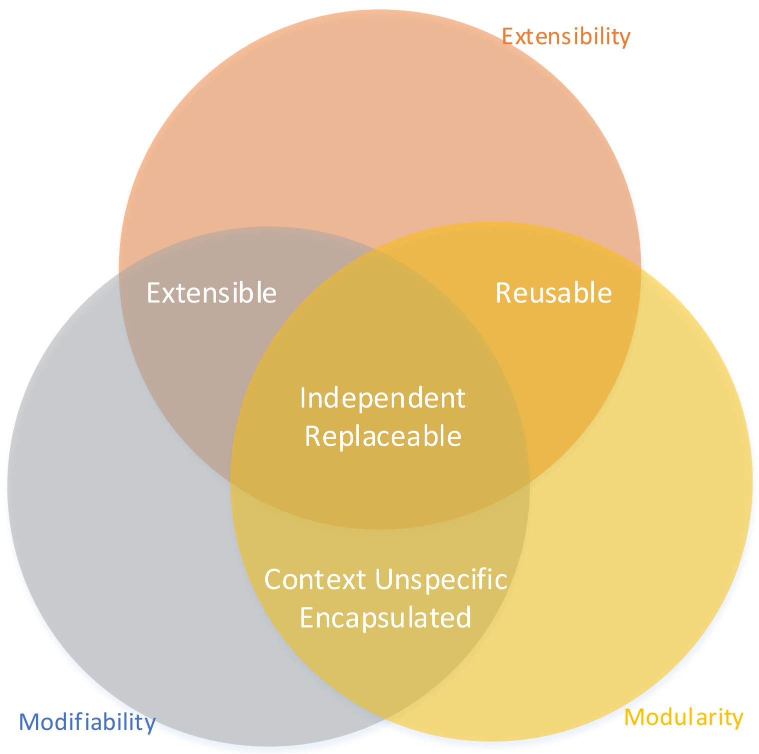


*Figure 18: Commonality between identified architectural drivers and core concepts of the "Component" architectural pattern*

The core advantages and principals of the Layered architectural pattern are [0]:

- Abstraction: This architecture provides an abstraction of the whole system without losing the definition of the relationships, roles and responsibilities of its layers.
- Encapsulation: Implementations details are not important to define as part of this pattern as properties such as data types, functions etc. are not exposed at layer boundaries. This is very important at the test components layer so we can expose a simple API to the test core layer, allowing for specific components to be plugged under the hood transparently.
- Clearly defined functional layers: A goal of the pattern, these functional layers should provide the implicit definition of relationships, roles and responsibilities that were mentioned above. This allows us to easily identify where packages belong and to understand what each layer’s purpose is.
- High Cohesion: By placing modules together in layers defined by relationship, role and responsibility you create a system which has a high level of cohesion.
- Reusable: Lower layers have no dependencies on higher layers, thus they are potentially reusable under different scenarios.
- Loose coupling: Abstraction provides loose coupling of communication between layers.
- Localised changes: This is one of the most important advantages for TOIT as any modifications or changes made are now confined to a layer, thus preventing a ripple effect.

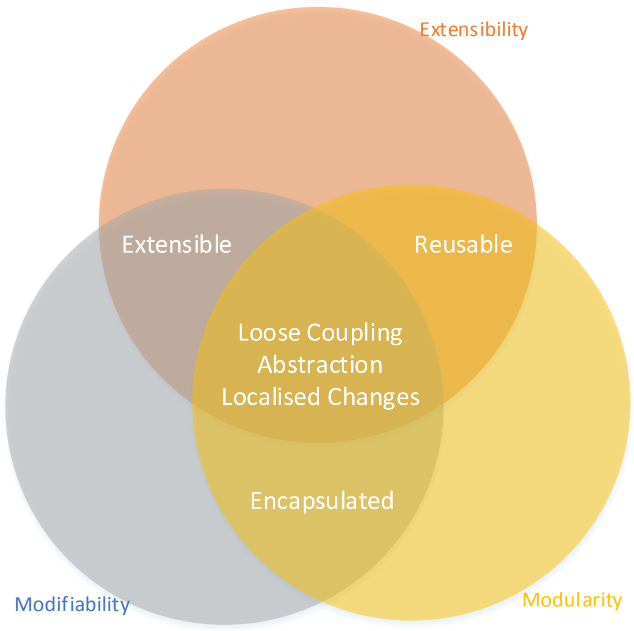


*Figure 19: Commonality between identified architectural drivers and core concepts of the "Layered" architectural pattern*

### Limitations and Risks

The limitations of the Component pattern are [1]:

- Performance: In some cases components are comprised of other components, which may in turn be comprised of other components again and so on, this pattern is not suitable for use in systems where performance at high speeds is critical as it can result in a lot of stack jumping. But as TOIT is mainly a configuration and collection tool, its performance is not a key concern.
- Code complexity: In order to create an object, its constituent objects must be created first. If the level of inclusion is deep, this can result in a lot of initialisation code. The trade-off between complexity and modularity is an acceptable one. The concern for TOIT could be minimised by ensuring this is an area of focus for code reviews for newly added components.

The limitations of the Layered pattern are [2]:

- Performance: Overhead of passing through layers can be detrimental to performance. As with the Component pattern, TOIT is mainly a configuration and collection tool, as such, its performance is not a key concern.
- Bubble up of changes: changes to lower levels which add new functionality will tend to bubble up to the upper layers as APIs extend to allow access to it. The majority of changes are expected to be additions of new implementations of vSwitches/VNFs/traffic generators etc. and as such they will not add new functionality, only enable existing functionality across new platforms. A risk area here is the addition of new test cases which would

require new functionality, but in this case the change is instigated from the top level, so it should be immediately apparent that the upper levels will change as a result.

## Quality Attribute Utility tree

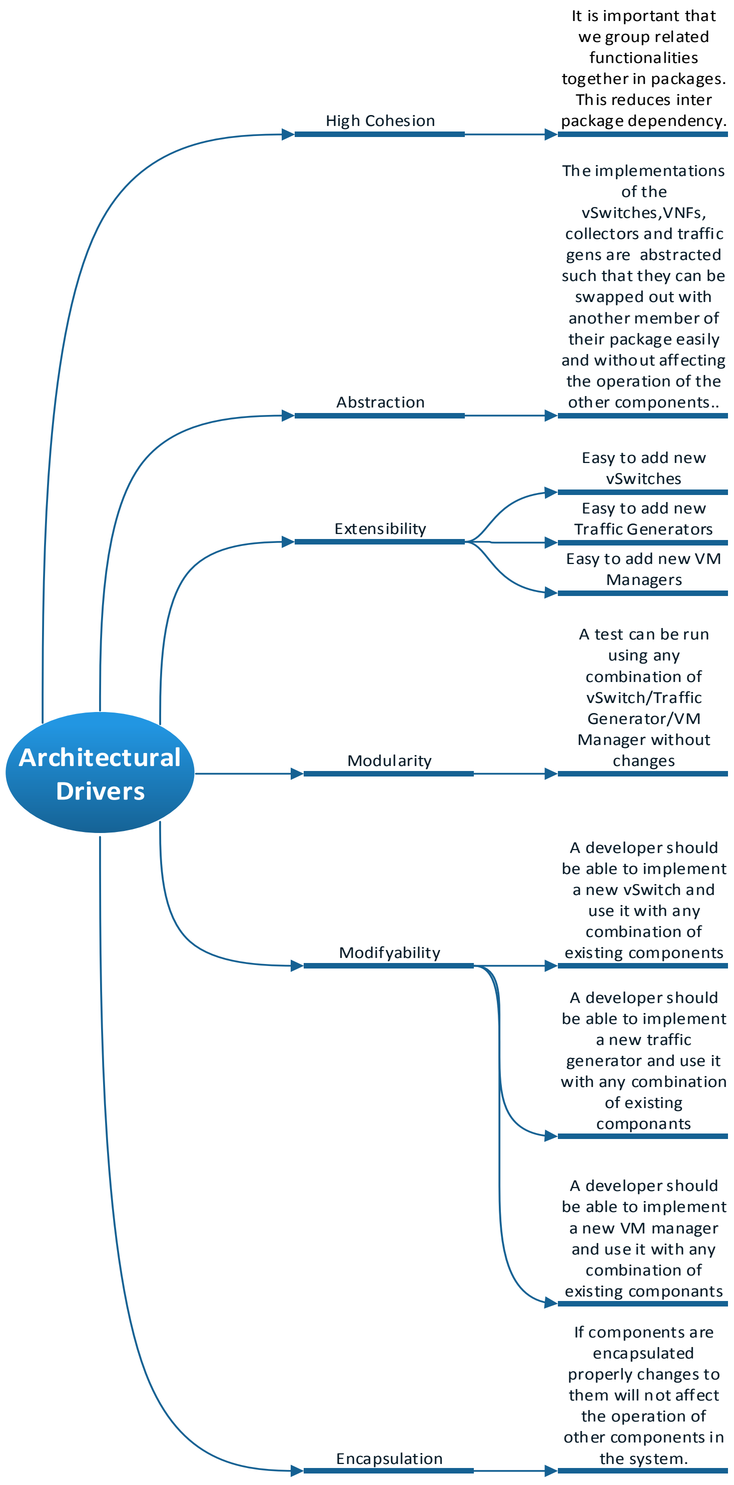

Architectural Drivers
High Cohesion
It is important that we group related functionalities together in packages. This reduces inter package dependency.
Abstraction
The implementations of the vSwitches,VNFs, collectors and traffic gens are abstracted such that they can be swapped out with another member of their package easily and without affecting the operation of the other components..
Extensibility
Easy to add new vSwitches
Easy to add new Traffic Generators
Easy to add new VM Managers
Modularity
A test can be run using any combination of vSwitch/Traffic Generator/VM Manager without changes
Modifyability
A developer should be able to implement a new vSwitch and use it with any combination of existing components
A developer should be able to implement a new traffic generator and use it with any combination of existing componants
A developer should be able to implement a new VM manager and use it with any combination of existing componants
Encapsulation
If components are encapsulated properly changes to them will not affect the operation of other components in the system.

*Figure 20 Quality Attribute Utility Tree*

## Addressing the Quality Attributes

Modularity, Modifiability and extensibility: will be achieved by decomposing the framework into individual packages that aim to abstract the complexity of underlying modules. Each module performs a very specific job and will expose a well-defined API for accomplishing their specific functionality.  For example, the vSwitches package will configure, run and clean up a vSwitch. This is the API it exposes to an upper layer that uses it. The specific virtual switch packages within vSwitches take care of the implementation details of what it means to configure, run and clean up a particular vSwitch.

High cohesion, abstraction, and encapsulation: will be achieved by placing modules that have related functionalities in a similar package and exposing a simple generic API to the upper layer that uses the package. The API hides the implementation details of the modules, for example the vSwitches API will expose an API function to add a port that takes an argument that is physical or logical. It’s up to the underlying vSwitch to determine what the meaning of a physical port is.