



The OVS Integration Test suite (TOIT) High Level Design

Document Number: 1.0

Document Revision: 0.0

Date: March 2015



Copyright Notice

Copyright (c) 2015 Intel Corporation and others.

All rights reserved. This program and the accompanying materials are made available under the terms of the Apache License, Version 2.0 which accompanies this distribution, and is available at <http://www.apache.org/licenses/LICENSE-2.0>



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

Key Contributors	First Name	Last Name	
	Stephen	Finucane	stephen.finucane@intel.com
	Michal	Weglicki	michalx.weglicki@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 a Telco NFV environments.

TOIT was extended as a starting point to help achieve the project goal. However, TOIT’s current architecture doesn’t expose generic enough APIs to test cases to allow for different vSwitch and traffic generators to be plugged in without test cases knowing a little about their implementation details. 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 Meghan Halton



Table of Contents

Approvals.....	iv
Abstract	1
Revision History	2
1. Introduction	5
1.1 Terminology.....	5
1.2 References	5
2. Assumptions, Dependencies	7
2.1 Assumptions	7
2.2 Dependencies	7
3. TOIT Architecture and Design.....	9
3.1 Aim.....	9
3.2 Overview	9
3.3 Current Architecture and Design.....	10
3.3.1 Architecture and Design Diagrams	11
3.4 TOIT Re-Architecture.....	13
3.4.1 Introduction	13
3.4.2 Architecture Trade-off Analysis Method (ATAM)	13
3.5 TOIT Redesign	29
3.5.1 Introduction	29
3.5.2 TOIT use cases.....	29
Implementation.....	35
Testing.....	35
Open Issues	35



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

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].





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 test specification.
- TOIT will expose generic APIs for test components such as vSwitches and traffic generators to allow for the integration with multiple traffic generator and virtual switch 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.

2.2 Dependencies

- “Characterize vSwitch Performance for Telco NFV Use Cases” Test Specification
- Existing TOIT
- DPDK
- OVS
- QEMU
- IXIA

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 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 (which will be referred to as 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 a 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
        |-- build_base_machine.sh
        \-- Fedora
        \-- Ubuntu
    \-- vswitches
        |-- add_switch
        |-- add_port
        |-- add_flow
        \-- ovs-dpdk
        \-- ovs-kernel
    \-- tools
        \-- pktgen
            |-- dpkt-pktgen
            |-- pktgen
            |-- pktcounter
            |-- spirent
            |-- ixia
        \-- collectors
    \-- testcases
        |-- p2p
        |-- pvp
        |-- pvvp
        |-- p2v
        |-- v2p
    \-- jobs
        |-- dpdk.conf
    \-- test_spec

```

- contains linux distributions
 - Input for generating Makefiles
 - Fedora specific setup
 - Ubuntu specific setup
- API to setup vswitches DUT
 - script to add switch
 - script to add ports on switch
 - script to add flow on switch
 - contains implementation on ovs-dpdk
 - contains implementation on ovs-kernel
- collections of tool sets
 - contains various packet generator
 - dpdk pkt generator
 - netmap pkt generator
 - a kernel based packet generator
 - script to control spirent
 - script to control ixia
- contains various data collectors
- collections of test cases
 - test PHY to PHY
 - test PHY to VNF to PHY
 - test PHY to VNF to VNF to PHY
 - test PHY to VNF
 - test VNF to PHY
- collections of job configurations
 - dpdk configuration
- contains test specifications

Figure 1: Characterize vSwitch Performance for Telco NFV Use Cases 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

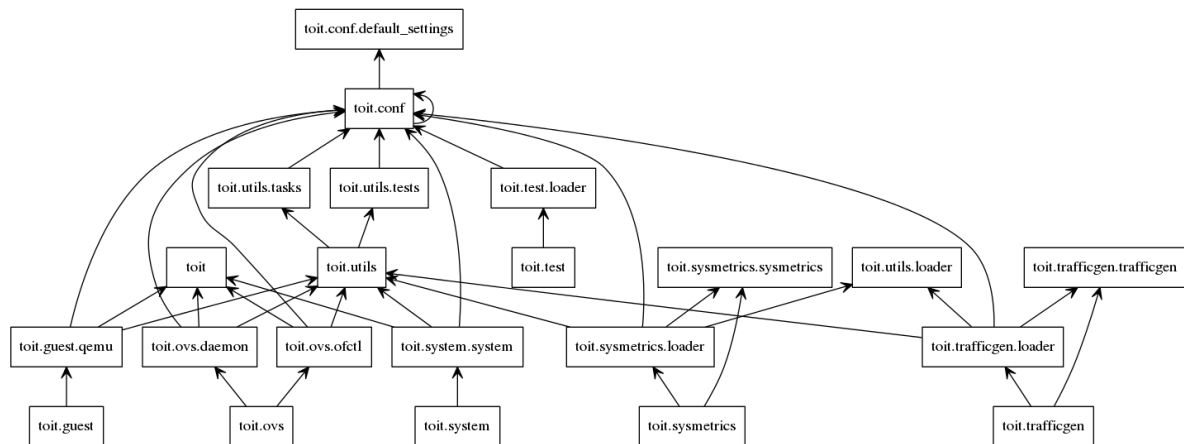


Figure 2: Package diagram of TOIT before rework

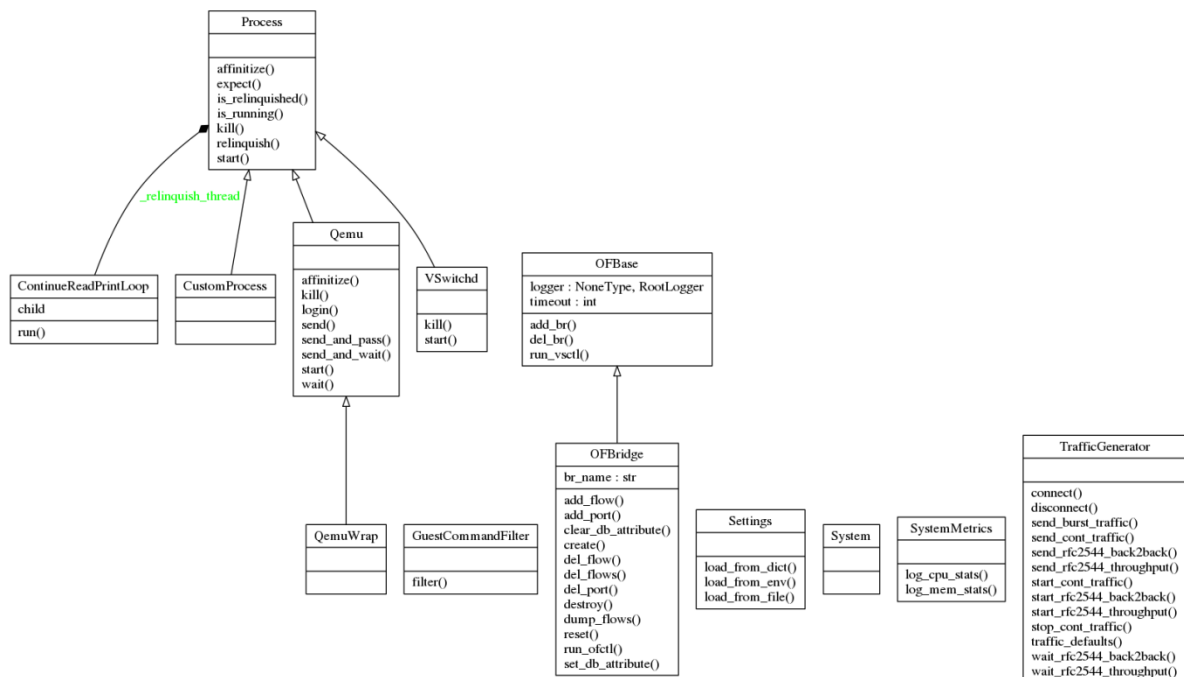


Figure 3: Class diagram of TOIT before rework

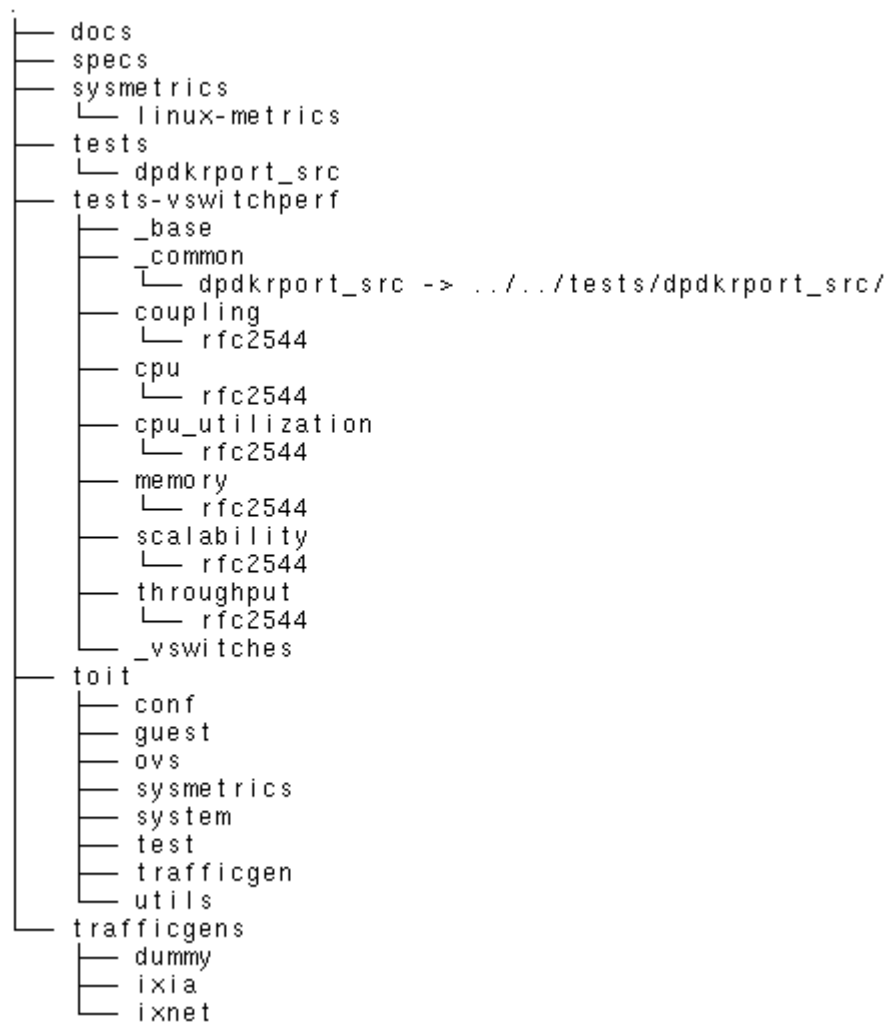


Figure 4: Current TOIT directory layout

3.4 TOIT Re-Architecture

3.4.1 Introduction

As a TOIT architecture design document did not exist 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.

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

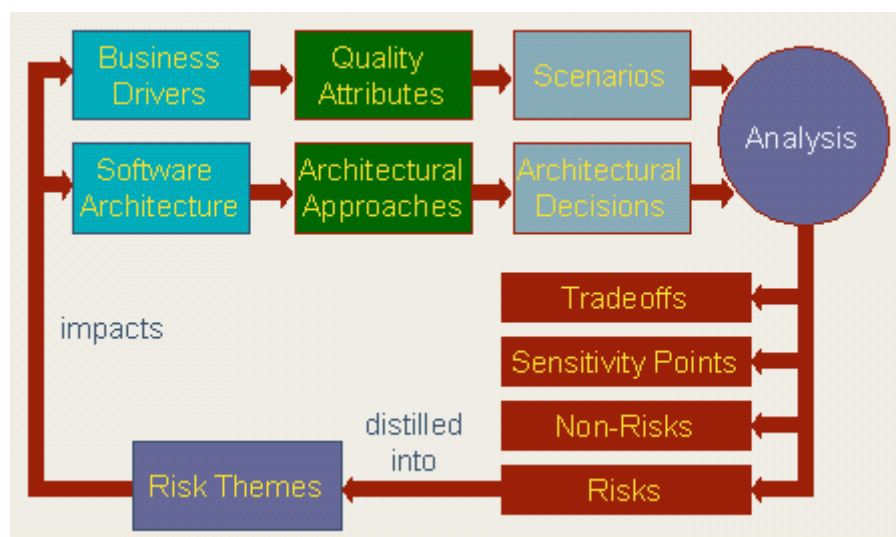


Figure 5 Conceptual flow of ATAM [4]

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 (cf. ATAM Summary).
- Presenting the system's business drivers (cf. 3.4.2.1). The business drivers help identify the system's quality attributes, which can be converted into specific use cases.
- Presenting the system's architecture (cf. 3.5). 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 (cf. 3.5) 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.

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 is an Apache licensed, python based, 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. It's part of the "Characterize vSwitch Performance for Telco NFV Use Cases" OPNFV project. This 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.

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 3.4.2 (released on October 8th, 2014). 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. TOIT also makes use of the Pylint and Tox modules to ensure code quality.

Targeted OS: CentOS 7 and Ubuntu 14.04

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:

The final step of the presentation phase prescribes providing 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 (cf. 3.3).

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

System diagrams

This section includes a structural and development view of the TOIT architecture. The structural view (cf. Figure 6) presents a high level overview of the proposed components for TOIT, which will be discussed in the section titled "Identify the architectural pattern". The development view (cf.

Figure 7) shows the packages that compose the various components; the internal structure of these packages will be discussed in section 3.5.

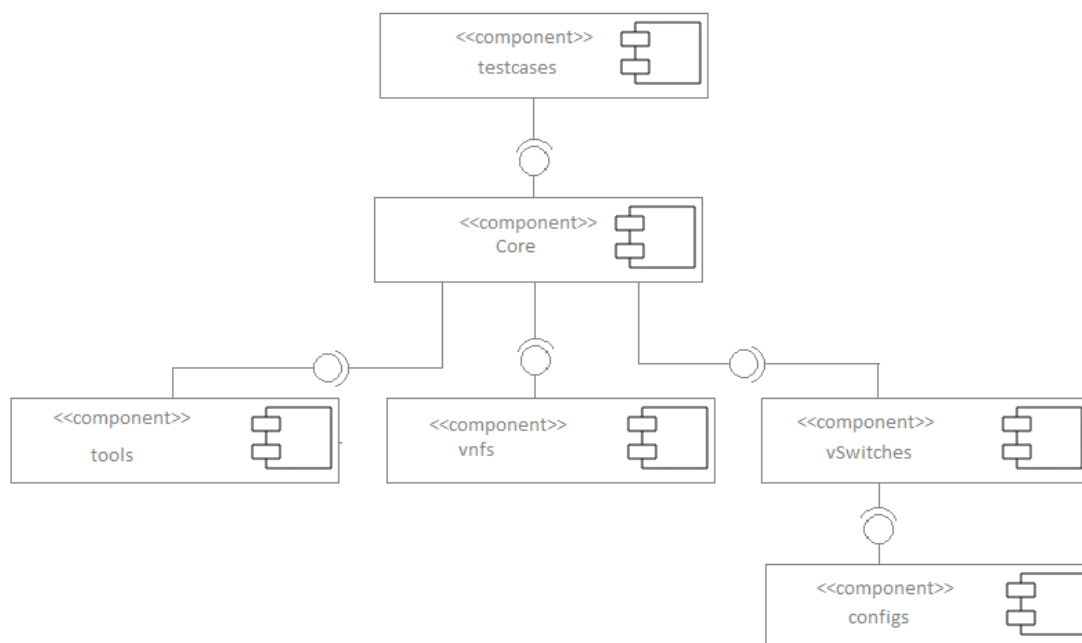


Figure 6 Structural view of TOIT

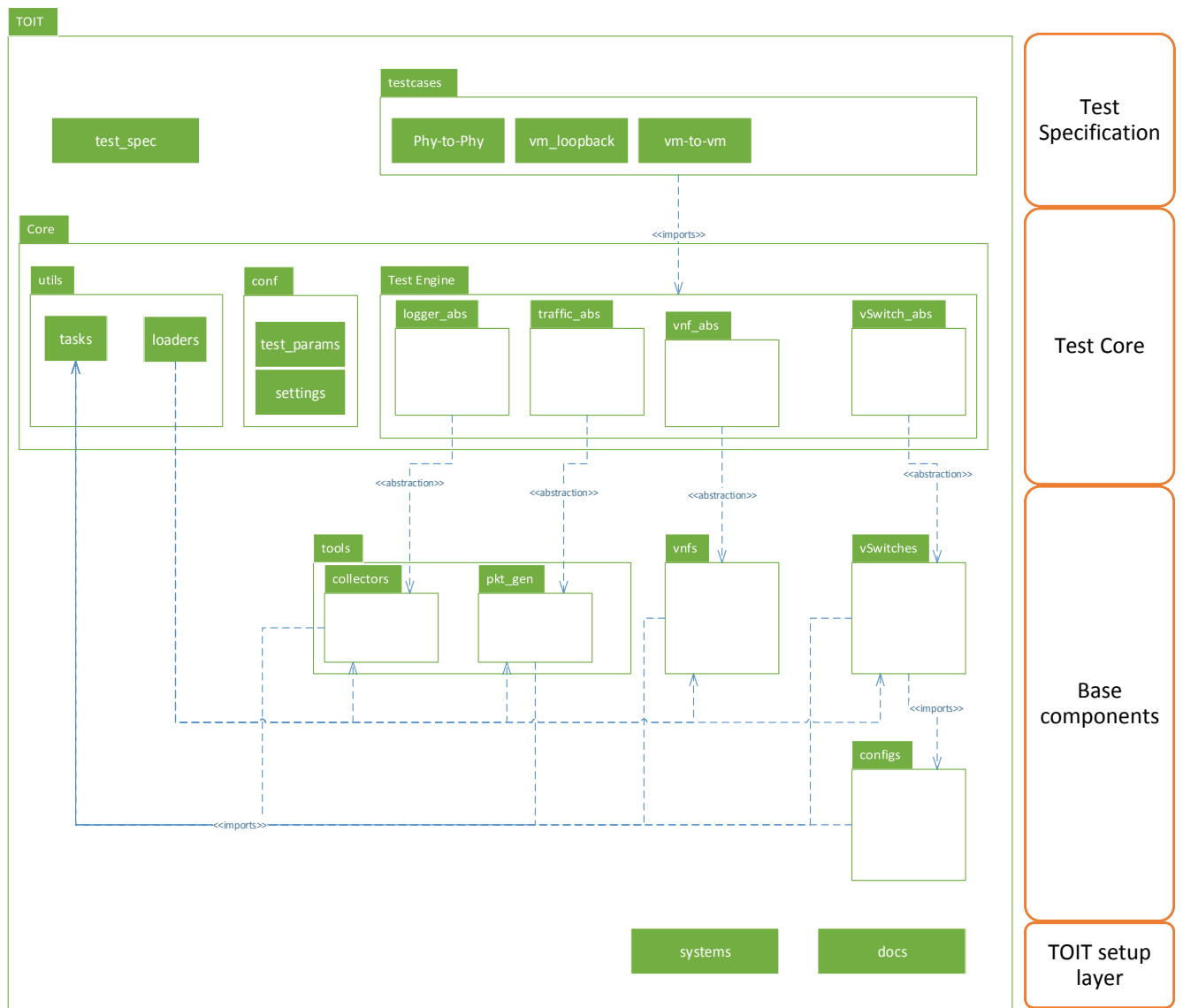


Figure 7 Development view of TOIT



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 identified in Figure 7 as: the test specification layer, the test core layer, the base components layer, and the TOIT setup layer.

Advantages 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.

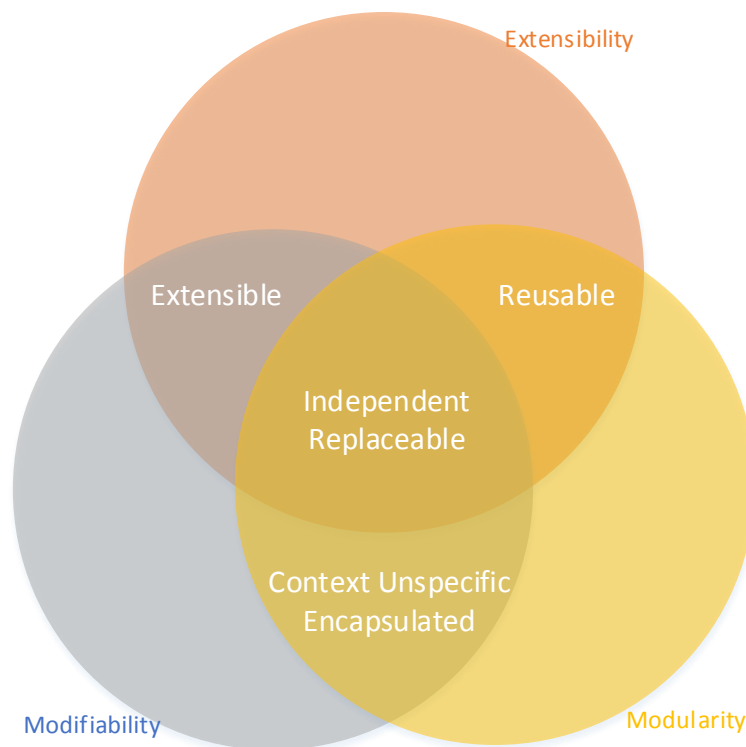


Figure 8: Commonality between identified architectural drivers and core concepts of the "Component" architectural pattern

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.

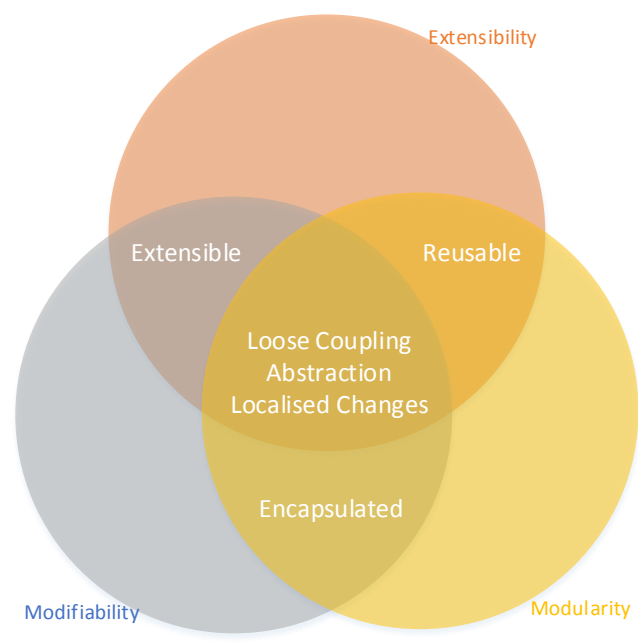


Figure 9: Commonality between identified architectural drivers and core concepts of the "Layered" architectural pattern

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

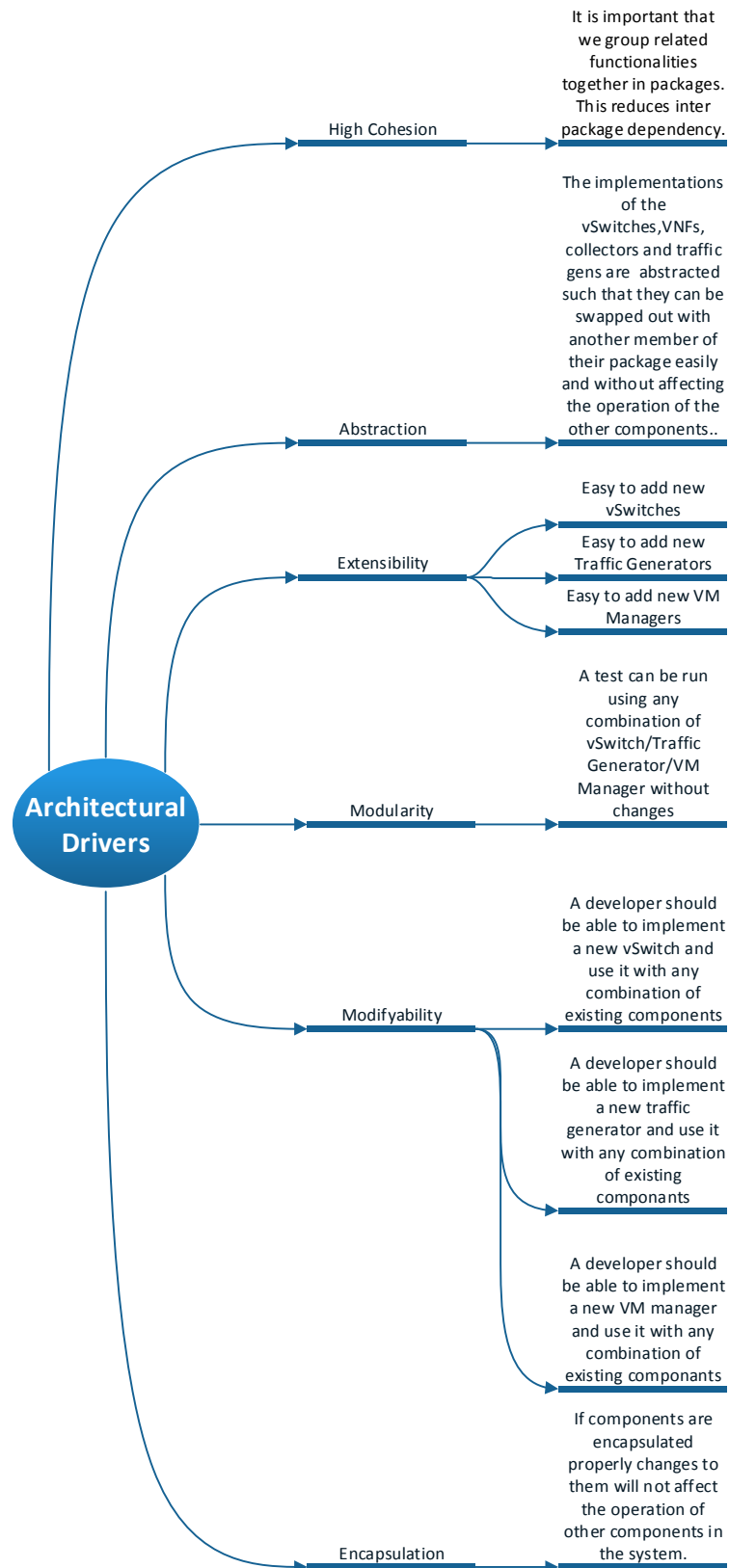


Figure 10 Quality Attribute Utility Tree

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.

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:
 - A vSwitch object from core.testengine.vswitch_abs.vswitch
 - A Traffic object from core.testengine.traffic_abs.traffic
 - 0 or more collectors objects from core.testengine.collector_abs.collector
 - 0 or more VNFs from core.testengine.vnf_abs.vnf



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

Testcase results (Traffic, Collector) will be written to a file by the testcase object itself, while the Traffic object and the collector object will manage the printing of results to screen. This is because there is no dependency on order when printing a result to screen, however when writing the result to one file, we need to make sure that results are output in the same order for each tests, traffic results first followed by logger results.

Test cases will retrieve their settings from `toit.core.test_params`.

Test case pseudo Code:

Test case x:

Get Settings

Create a new vSwitch: `vSwitch_abs("P2P")`

Create a new traffic object: `traffic_abs("RFC2544")`

Send traffic: `traffic.send ()`

Get results: `traffic.get_results()`

`write_results_to_file()`

- core: The purpose of this package is to hide the implementation details of the test components layer from the test specification 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.

Package abstractions at this layer are not aware of the implementation details of packages at the test components layer. For example `core.testengine.vswitch_abs` class/object will use a simple API to add a port to a vSwitch `add_port(string port)`,

where port can be phy or logical. It's up to the underlying vSwitch in the vSwitches package to determine what a phy 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_abs to setup a p2p vSwitch, the vSwitch_abs 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 to collect CPU usage and memory usage information) and traffic generators.
- configs: the purpose of this package is to configure and setup a system with the dependencies for a test case. It's preferred that if a 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 switch 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	<p>This package (as well as its sub packages and modules) will be moved to the tools.collectors package under the new directory structure.</p> <p>The modules in this package will implement a new function collect (type), where type can be mem_stats or cpu_stats or other.</p>
toit.tests-vswitchperf	This sub packages and modules in this package will be moved to the testcases package in the new directory structure.



	Testcases will be organized based on deployment scenario. Any Testcase which relies on the results of a previous Testcase will test for those results/run the new Testcase if those results are not present.
toit.trafficgens	This package (as well as its subpackages and modules) will be moved into tools.pkt_gen in the new directory structure.
toit.toit	<p>toit.toit.conf will move to core.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.</p> <p>toit.toit.ovs and its modules will move to vswitches.ovs_dpdk and will implement a generic vSwitch API that will allow the core.testengine.vswitch_abs 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.testengine.vswitch_abs) vSwitches, ports and flows are abstracted into simple objects to hide the underlying complexity. For Example a core.testengine.vswitch_abs 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.</p> <p>vswitches.ovs_dpdk will also look after configuring DPDK for OVS.</p> <p>toit.toit.guest will move to vnfs.qemu</p> <p>toit.toit.sysmetrics will be removed as it's now realized as part of tools.collectors.sysmetrics.</p> <p>toit.toit.system was configuring and setting up DPDK for OVS. toit.toit.system.system will be moved under configs and renamed to dpdk_config.py. The</p> <p>toit.toit.test.loader will be moved to the core.utils.loader package.</p> <p>toit.toit.trafficgen will be moved into tools.pkt_gen package.</p> <p>toit.toit.utils.loader please see toit.toit.utils.loader row.</p> <p>toit.toit.utils.tasks will be moved to core.utils.tasks.</p> <p>toit.toit.utils.tests please see toit.toit.utils.tests row.</p>

toit.toit.trafficgen.loader toit.toit.sysmetrics.loader toit.toit.utils.loader	<p>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 a loaders package under core (core.loaders). The function will take the path as an argument.</p>
toit.toit.utils.tests	<p>The old module toit.toit.utils.tests simply contains a function to get test parameters, this module will be moved to core.conf as this package contains the default test parameters and the functions pertaining to retrieving them. The modules will be renamed test_params and will implement get_test_param.</p>
toit.toit.conf.Default_settings.py	<p>Default_settings.py will be broken up into several configuration files for TOIT settings, vSwitches, VNFs and traffic generators.</p> <p>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.</p> <p>The user will be able to override configurations by modifying a user_conf.conf file.</p> <p>The settings module will be accessed by nearly every package in the system. This is because it avoids the upper module having to worry about what settings structures it needs to pass down to lower levels.</p>
	<p>Jobs package/directory will be renamed to configs, as its naming should reflect its purpose which is to configure any dependencies a vSwitch or other components may have. This package is a collections of configurations from different users. It is not the configuration for the test framework to run.</p>

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.

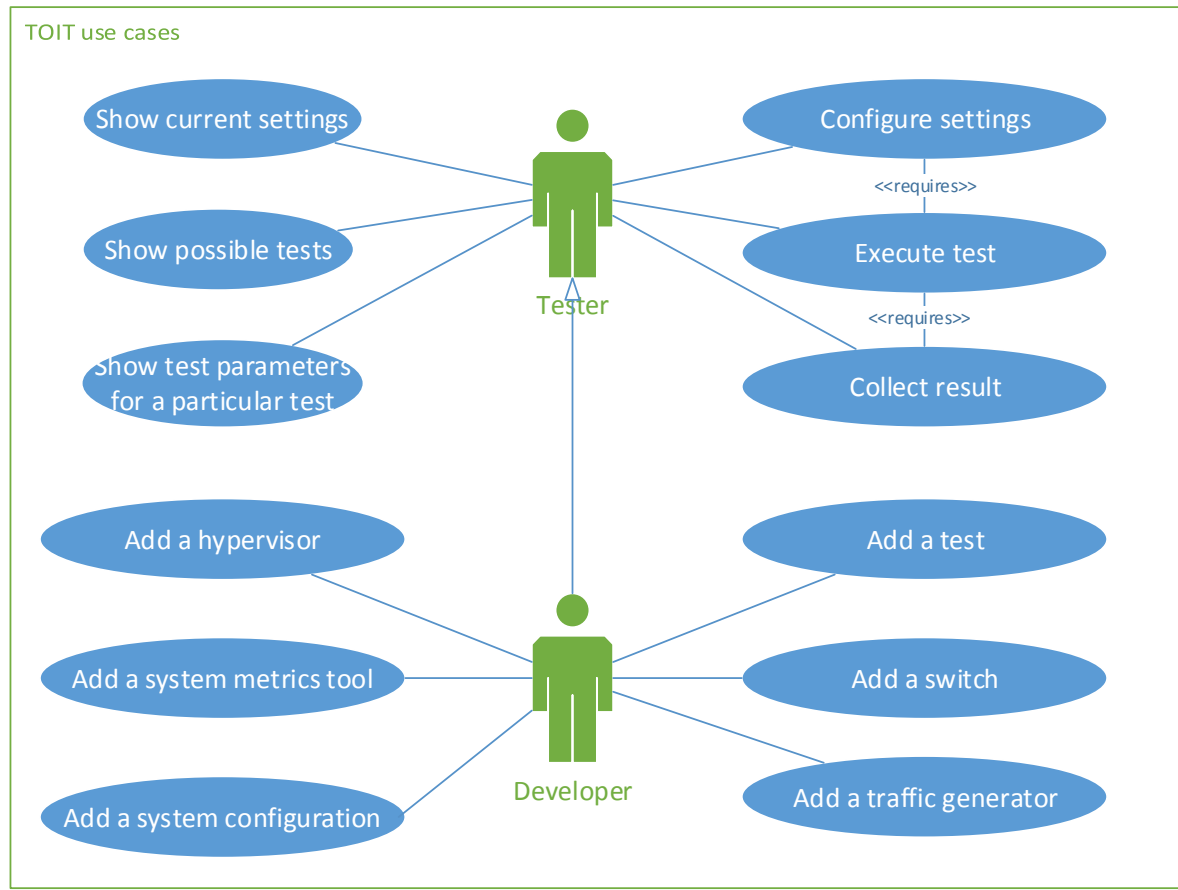


Figure 11: TOIT use cases

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	<ol style="list-style-type: none"> 1. Tester receives system 2. Tester runs a command to output the current settings profile 3. Tester modifies the system (by selecting for instance another traffic gen). 4. 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	<ol style="list-style-type: none"> 1. Tester runs a command to list tests. 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	<p>Test parameters exist for a particular test.</p> <p>The Tester knows the name of the test they wish to call.</p>
Post conditions	The test parameters for the listed test have been output to screen.
Flow of activities	<ol style="list-style-type: none"> 1. The Tester selects their required test. 2. The Tester runs a command to print the test parameters for the selected test. 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 the parameters and settings used for running the tests.
Actors	Tester, Developer
Preconditions	A default configuration file exists
Post conditions	New settings are defined in a configuration file.
Flow of activities	A developer or tester creates or modifies the settings.py file.
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	<ol style="list-style-type: none">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.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	Collect result
Description	As a tester I would like to collect the results of a test that was run.
Actors	Tester, Developer
Preconditions	A test was run.
Post conditions	A results report is available.
Flow of activities	<ol style="list-style-type: none">1. The tester or developer runs the test by test name or category.2. The tester or developer collects 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	<ol style="list-style-type: none"> 1. Developer creates new file or modifies an existing file within testcases/(deployment scenario) based on their deployment scenario 2. Developer implements abstract methods defined by the Testcase interface class. 3. Developer adds the necessary configuration to the default configuration file. 4. Developer includes either a README with their new implementation or updates an existing document.
Exception condition	The Developer wishes to add a test 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 tests, before implementing their new test in a separate patch.

Use Case Number	8
Use Case Name	Add a switch
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	<ol style="list-style-type: none"> 1. Developer creates new folder within vSwitches/ 2. Developer creates a file in that folder. 3. Developer implements abstract methods defined by the vSwitch interface class. 4. Developer adds the necessary configuration to the default configuration file. 5. Developer includes either a README with their new implementation or updates an existing document.



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	<ol style="list-style-type: none">1. Developer creates new folder within tools/trafficgens/2. Developer creates a file in that folder.3. Developer implements abstract methods defined by the Trafficgen interface class.4. Developer adds the necessary configuration to the default configuration file.5. Developer includes either a README with their new implementation or updates an existing document.
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 hypervisor
Description	As a developer, I would like be able to add support for a hypervisor so that a tester can bring VMs up for testing VM to VM and VM loopback traffic, without affecting any existing implementations of other hypervisors.
Actors	Developer

Preconditions	Modular implementation of hypervisors. Single API that all hypervisors conform to.
Post conditions	New hypervisor definition which allows the use of a new type of, or version of, hypervisor.
Flow of activities	<ol style="list-style-type: none"> 1. Developer creates new folder within tools/hypervisors/ 2. Developer creates a file in that folder. 3. Developer implements abstract methods defined by the hypervisor interface class. 4. Developer adds the necessary configuration to the default configuration file. 5. Developer includes either a README with their new implementation or updates an existing document.
Exception condition	The Developer wishes to add a hypervisor 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 hypervisor in a separate patch.

Use Case Number	11
Use Case Name	Add a system metrics tool
Description	As a developer, I would like to be able to add support for a system metrics logger so that a tester can use a different system metrics logging tool depending on the degree of accuracy or type of metric they would like to gather. This should be done without affecting the operation of any existing system metrics loggers.
Actors	Developer
Preconditions	Modular definition of a system metrics provider. Single API that all system metrics loggers conform to.
Post conditions	New system metrics provider implementation which allows the use of a new type of system metrics logger.
Flow of activities	<ol style="list-style-type: none"> 1. Developer creates new folder within tools/sysmetrics/ 2. Developer creates a file in that folder. 3. Developer implements abstract methods defined by the Sysmetrics interface class. 4. Developer adds the necessary configuration to the default configuration file. 5. Developer includes either a README with their new implementation or updates an existing document.
Exception condition	The Developer wishes to add a sysmetrics 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 sysmetrics



	providers, before implementing their new sysmetrics provider 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	<ol style="list-style-type: none">1. Developer creates new folder within /systems/2. Developer creates a file in that folder.3. Developer implements abstract methods defined by the System interface class.4. Developer adds the necessary configuration to the default configuration file.5. Developer includes either a README with their new implementation or updates an existing document.
Exception condition	The Developer wishes to add a system configuration 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 system configuration providers, before implementing their new system configuration provider in a separate patch.

TODOs:

- Class diagrams.
- Sequence diagrams
- Directory Tree.

Implementation

Testing

Open Issues