Test and Simulation Toolkit (TSTK) Documentation

Release 0.0.1

F.M. Walinga

March 27, 2014

CHAPTER

ONE

USER MANUAL:

1.1 Introduction

This document is the Software User Manual (SUM) for the Test and Simulation Toolkit (TSTK). This SUM will explain the possibilities of the TSTK and how to use it. This manual is intended for both InTraffic employees and developers from the open source community.

This manual can contain some basic explanation about Git(hub) workflow and/or procedures. The reason behind this is because not everyone might be familliar with them.

1.2 Software Summary

1.2.1 Software-application

This Test and Simulation Toolkit, from this moment on called TSKT, is a combination of an install script and a package containing drivers. With this the TSTK it's possible to install an existing test system, but only if it has been created using the TSTK. It's also possible to create an environment in which you can develop a new test system. The TSTK also provides the possibility to update the test system or developing environment.

With the TSTK the need to create specialized drivers for each test system is limited to the translation of messages that are to be sent over a connection to your test system. Besides this, the installation, distribution and updating of test systems is greatly simplified by just executing one script which will do the work for you.

When you have created a test system with the TSTK you can create a package for distribution and installation on other systems. This will be explained in detail in chapter 5.

1.2.2 Software-inventory

The TSTK is created to be used on Linux systems. The software that must be present on the system for the TSTK to function is:

- Python 2.7
- Python 3.2
- Python-virtualenv
- Git

1.2.3 Software-environment

The environment, in which the TSTK will run, shall have to be composed of computer with the following recommended specifications:

- Ubuntu 12.04 or similar OS
- 2GHz Dual core
- 2 GB RAM
- 500 MB of free disk space
- The system is connected to the internet

1.2.4 Software organization and overview of operation

The TSTK contains the following two components:

- An install script. This script creates the environment and installs all the drivers.
- A package containing all drivers for the test system or the test system itself. The contents of the package depend on whether you are installing an existing test system or if you are installing an environment to develop a new test system.

1.2.5 Contingencies and alternate states and modes of operation.

The installation script will keep a log during the installation. When something does not execute as it should, the installation is aborted and everything will be reversed. This will al be logged.

1.2.6 Assistance and problem reporting

For assistance and problem or bug reporting you should go to the GitHub page at https://github.com/InTraffic/TSTK .

Problems and bugs should be reported by creating a new Issue, describing the problem and with the correct label. This can be found at https://github.com/InTraffic/TSTK/issues

For assistance you should check if it hasn't been mentioned in this document. If it hasn't, you should go to the wiki on the GitHub page at https://github.com/InTraffic/TSTK/wiki. If that also doesn't solve the problem, you should create a new issue and attach the "question" label to it.

1.3 Access to the software

This chapter assumes that the software from paragraph 3.2 has been installed on the computersystem and that the environment meets the specified requirements.

1.3.1 First time user of the software

Equipment familiarization

To access the TSTK you will have to download the install script from the GitHub page. When you have downloaded the script, you can execute the following steps to show the possibilities:

- Open a new terminal window.

- Navigate to the folder where the install script has been downloaded, or moved, to.
- Type this command: `python2.7 TSTK-install.py --help`
- Hit the enter key to execute the command
- You will now be shown the possible options and argument you can pass to the install script.

The possible options and arguments are displayed below:

```
Test and Simulation Toolkit installer script

Usage:
installscript.py install (existing TESTSYSTEM|
                          development TOOLKIT_PACKAGE)
                         [options...] FOLDER
installscript.py update FOLDER (PACKAGE...)
installscript.py -h | --help

Arguments:
 TESTSYSTEM This is the package containing the testsystem that is to be
            installed
 FOLDER     This is the folder that'll be created for the testsystem to
            reside in. When updating this is the folder containing the
            current testsystem.
 TOOLKIT_PACKAGE
            The package to use when setting up the environment for
            development.
 PACKAGE    The package to update to.

Options:
 -l <package>, --local-packages <package>
            Use this when you want to  install additional
            packages from the current folder. The input should
            be a tarball.
 -e <package>, --external-packages <package>
            Use this when you want to install additional
            packages from PyPI.
 --no-virtualenv
            Using this option will result in an OS wide install of the
            toolkit. It is NOT recommended to use this argument unless
            you know what you're doing and are willing to risk
            destabilizing your current package "ecosystem".
            Use at your own risk!
```

Installation and setup

The install script has to be downloaded, that's it. The directory it resides in should allow for executing programs,though.

1.3.2 Initiating a session

Initiating the installation of a development environment

The initiation of the installation of a development environment is as follows:

```
Python2.7 TSTK-installscript.py install development [toolkit_package] [folder]
```

[toolkit_package] should be substituted with the package containing the drivers. Either a path (eg. /foo/bar/baz.tar.gz), a relative path (eg. foo/baz.tar.gz) or a package name (eg. baz.tar.gz or baz) is accepted. When only the package name is provided, without the .tar.gz file extension, the TSTK will get the package from the Python Package Index.

[folder] should be substituted with the folder you want to create for the environment to reside in. Either a path (eg. /foo/bar/baz), a relative path (eg. foo/bar) or a foldername (eg. baz or baz/) is accepted.

The result of the above command will be a folder, with the specified name, containing a virtual environment with the default driver package installed. There will also be a Git repository initiated in this folder, since a development environment is set-up.

Initiating the installation of an existing test system

The initiation of the installation of an existing system is similar to the installation of a development environment, it is as follows:

```
Python2.7 TSTK-installscript.py install existing [package] [folder]
```

[folder] should be substituted with the folder you want to create for the environment to reside in. Either a path (eg. /foo/bar/baz), a relative path (eg. foo/bar) or a foldername (eg. baz or baz/) is accepted.

[package] should be substituted with the tarball of the test system package you want to install. Either a path to a package, a relative path to a package or a just a package is accepted.

The result, of the above command, will be a folder with the specified name. It will contain a virtual environment, and the testsystem installed within.

Initiating the update of a test system or development environment

To initiate the update of an existing system you use the following command:

```
Python2.7 TSTK-installscript.py update [folder]
```

[folder] should be substituted with the folder where the testsystem or development environment resides in. It will update the default set of drivers for you.

Initiating the installation and using the options

The install script accepts three options you can use when running the script.These options are:

- local-packages. This option enables you to specify a tarball, specifically a python package, that is installable through PIP. This tarball will then be installed in the virtual environment by the install script.
- external-packages. This option enables you to specify a package on PyPI. This package will be downloaded and installed in the virtual environment by the install script.
- no-virtualenv. This option will cause the installation script to install all pakages in the system-wide environment. WARNING: This can cause unforseen problems and stability/dependency issues. *Use this only when you know what you are doing and at you own risk!*

An example of the syntax for the use of local-package or external package is: Python2.7 TSTK-installscript.py install development –l Foo.tar.gz /foo/bar/baz To install multiple local packages you have to repeat te option, like: Python2.7 TSTK-installscript.py install development -l foo.tar.gz –l bar.tar.gz baz/

1.3.3 Stopping and suspending work

To stop the install script during the installation just hit ctrl + C. The installation will be aborted and the changes will be reverted.

1.4 Driver Module reference guide

1.4.1 Software reference guide

Capabilities

The driver module contains all drivers developed for use with the TSTK. The driver module contains a method which will allow you to get the desired driver.

How to use the driver module

To use the driver module you import the module in your program, like:

```
Import driver
```

To get a new object from one of the available drivers, you should use the get_driver function. This function takes two arguments, a name and a driver_id. An example of this is:

```
portal_driver = driver.get_driver("portal", 1)
```

This example will return a new instance of `driver.Portal` You can then use portal_driver by doing:

```
fetched_page = portal_driver.fetch("http://www.google.com")
print(fetched_page)
```

1.5 Simulator Module reference guide

1.5.1 Capabilities

The simulator module contains multiple kinds of Simulators. The Simulators consist three parts:

1. A daemon
2. A dispatcher
3. A message

The simulator module contains an AbstractFactory which will allow you to get the desired concrete Daemon and Dispatcher. The specific message has to be created by the developers that use the TSTK to create a new Testlab. The location of the message.py file that contains the specific message has to be specified in the `simulator.conf` file.

1.5.2 How to use the Simulator module

Creating a Message class

You have to create a message class if you want to use this module. The this message class will need to translate the messages you want to send to the system that is being tested. It also has to translate the messages you receive from

that system, so it can be sent back to the simulator interface. There are two methods that will be called:

- `to_message(command)`. This will process the command from the testsystem and must return the message.
- `from_message(message)`. This will process the message from the system you are testing and must return the command to send to the testsytem.

Configuring your simulators

To configure your simulator yo uneed to specify some things for the dispatcher part in the `simulator.conf` file. The config files are parsed by the configparser module from the python libraries. An example of this is:

```
[dispatcher-tcp-2]
MessagePath=/home/foo/bar/
AcceptAddress=127.0.0.1
ListenPort=9010
CommandListenPort=9000
MessageForwardPort=9001
```

The entries in the config file that the dispatchers look for are:

- MessagePath
- CommandListenPort
- MessageForwardPort

The dispatchers, excluding the serial dispatcher, also look for the following entries:

- AcceptAddress
- Listenport

Starting a simulator

Starting a simulator is quite a simple thng to do. You import the simulator module with `import simulator`. Then you just excecute the `simulator.start_simulator("simulator_type", simulator_id)` function with the correct simulator type and id as arguments.

The simulator module will start a new simulator of the specified type and with the given id. It will use this type and id to look for the config entries as specified in the above "Configuring your simulators" piece.

1.6 Test System Module reference guide

1.6.1 Capabilities

The Test System module is the core of the Testlab you can create with the TSTK. It consists of three parts:

- The testsystem class.
- The scenario player.
- Some simulator interfaces.

The testsystem class is the part where testscripts will interact with the Testlab. Testscripts will tell the testsystem what drivers, simulatorinterfaces, steps to add and to play or stop playing the steps. The scenarioplayer stores all steps for a test and executes them at the right time.

The simulatorinterface(s) are for communicating with the running simulator(s), they have to be implemented by the developer(s) who create a Testlab. A simulatorinterface has to start the correct simulator through the simulator module within the TSTK. The simulator sends commands to the simulator and can receive replies.

1.6.2 How to use the Test System module

Creating a simulatorinterface

To use a simulator interface, a developer for a Testlab must create an implementation of a specific simulator interface. If desired it can use the standard simulatorinterface class as superclass and inherit the attributes and methods from it. Be sure to add the new simulatorinterface to the list in the `get_simulator_interface` function in simulatorinterface.

Using the testsystem

To use the testsystem, you import it in your testscript with `from testsystem import testsystem`. You can then instantiate a new object from the TestSystem class to access the different methods as described in the "Software Documentation" part of this guide.

1.7 Expanding the TSTK

This part will explain what to do if you want to extend or expand the TSTK. An example of extending/expanding is adding a new driver to the driver module.

1.7.1 Part one: Github

If you want to add something to the driver, you should first have some knowledge of Github and the workflow. I recommend you to go to http://guides.github.com/ and read the guide if you aren't familiar with it.

1.7.2 Part two: The modules

The TSTK contains three modules:

- The testsystem module
- The driver module
- The simulator module

The driver and simulator modules are the modules where the most extending/expanding is expected.

Adding to the driver module

To add to the driver module you should add the new class to the driver.py file in the driver module folder. After that you should also add the reference to the `get_driver` function in the same file.

Adding to the simulator module

To add to the simulator module you have to add a daemon and dispatcher to their respective files in the simulator module folder. You should also create the corresponding connectionfactory and entry in the list with known factories in the abstractconnectionfactory.

1.8 Designs for the TSTK

1.8.1 Component Diagram

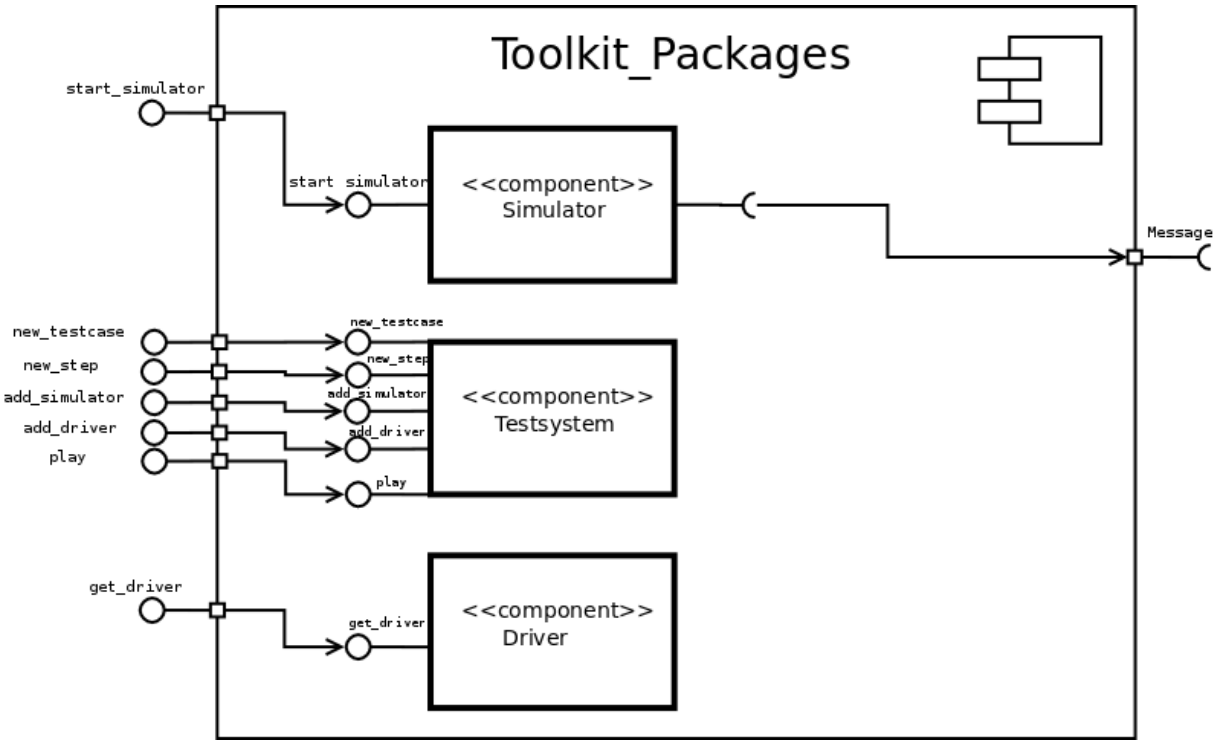

Figure 1.1: Component diagram for the Toolkit packages

1.8.2 Simulator Class Diagram

1.8.3 Test System Class Diagram

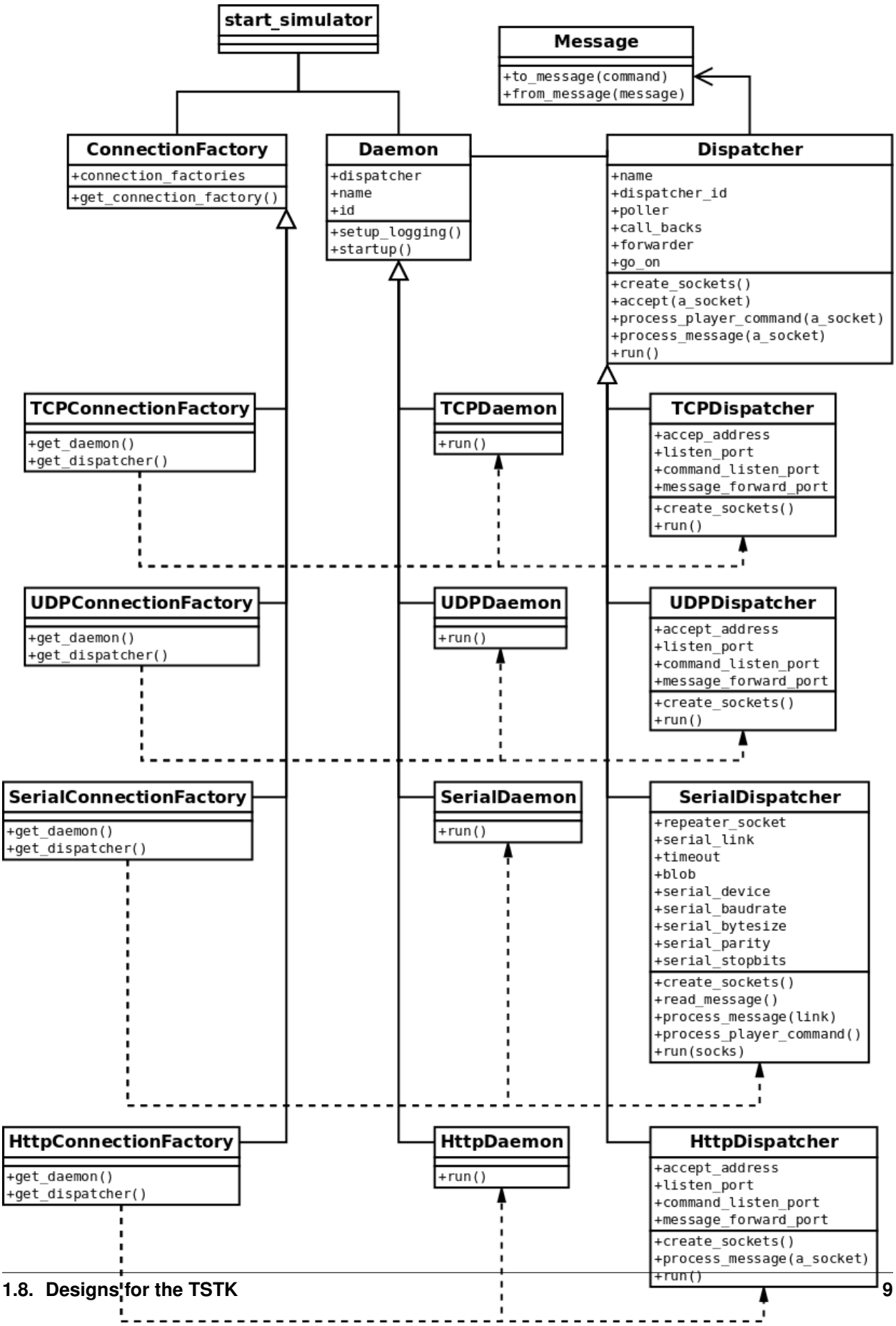

Figure 1.2: Class Diagram for the Simulator module

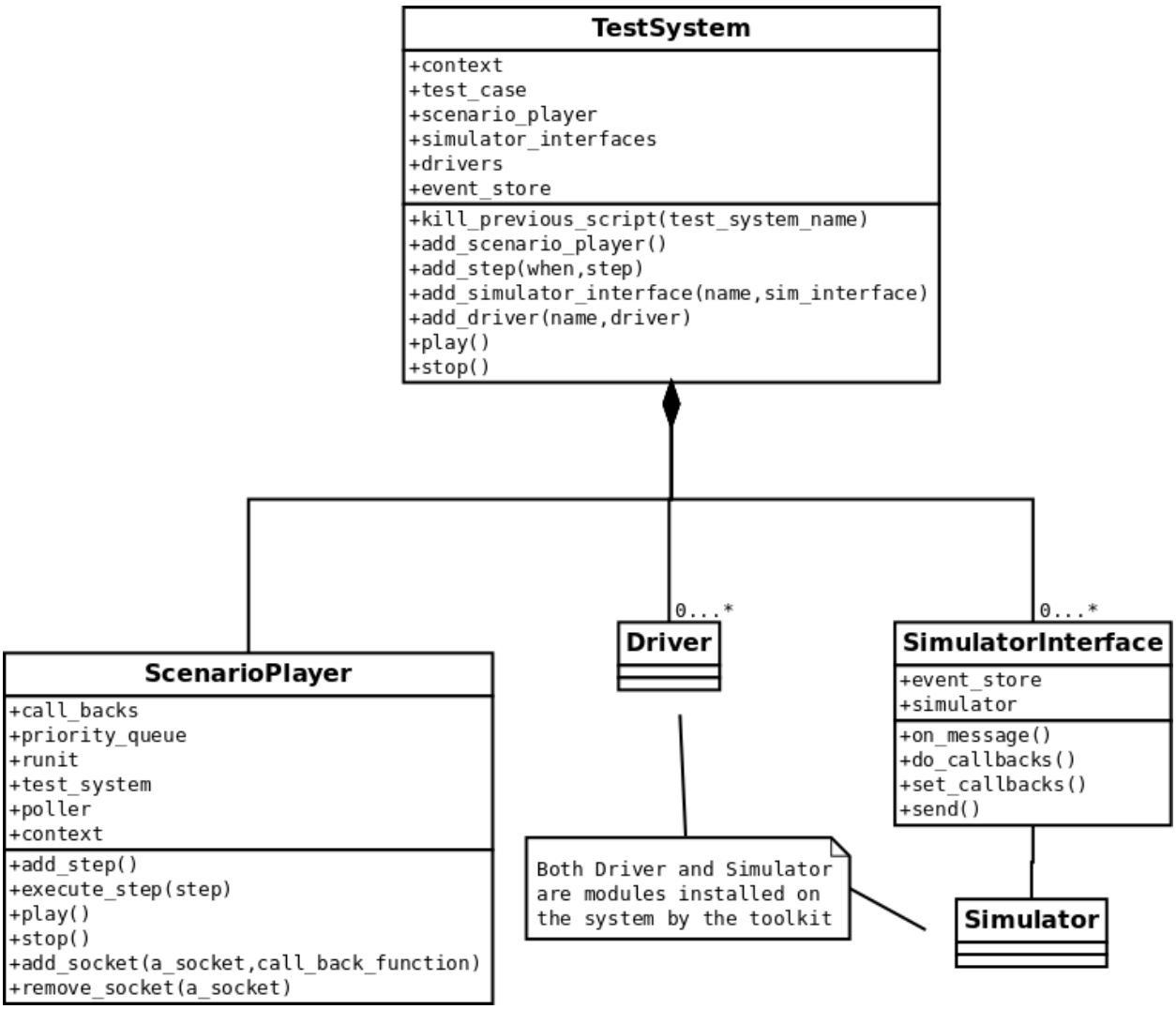

Figure 1.3: Class Diagram for the Test System module

CHAPTER

TWO

SOFTWARE DOCUMENTATION:

2.1 Driver Module

2.1.1 Software reference guide

Capabilities

The driver module contains all drivers developed for use with the TSTK. The driver module contains a method which will allow you to get the desired driver.

How to use the driver module

To use the driver module you import the module in your program, like:

```
Import driver
```

To get a new object from one of the available drivers, you should use the get_driver function. This function takes two arguments, a name and a driver_id. An example of this is:

```
portal_driver = driver.get_driver("portal", 1)
```

This example will return a new instance of `driver.Portal` You can then use portal_driver by doing:

```
fetched_page = portal_driver.fetch("http://www.google.com")
print(fetched_page)
```

The Portal driver will remember the url you wanted to fetch, so next time you can just use: `fetched_page = portal_driver()`

If you haven't passed an url through `driver.Portal.fetch()`

2.1.2 Get driver function

driver.**get_driver**(*name*, *driver_id*)
function to get the desired driver object from this module.

Parameters

- **name** (*string*) – name of the driver (eg. portal or usbrly08b)
- **driver_id** (*int*) – The id for the driver

2.1.3 Portal Driver

class driver.**Portal**(*portal_id=None*)
Interface to a web portal

fetch(*portal_url=None*)
Fetch the current version of the web portal.

The Portal driver will remember the url you wanted to fetch, so next time you can just use: `portal_driver()` If you haven't passed an url through `driver.Portal.fetch()` the function will return 1.

Parameters **portal_url** (*string*) – the url to fetch, this url will be remembered untill the next call with a portal_url specified.
Returns The fetched web portal if an url is known or specified
Returns 1 if no url is known or specified

2.1.4 Usbrly08b Driver

class driver.**Usbrly08b**(*device_id*, *debug=False*)
This class is an interface to the USB-RLY08 serial device.

This class can be use directly through the open_relay and close_relay methods. It is recommended to use this class as a superclass for an implementation of a more descriptive subclass in a project. This'll allow for the addition of logging and better understandable methods and/or functions.

Attributes: device_id – the id of the usbrly08b board to communicate with. debug – whether or not this module is to run in a debugging mode.

close_relay(*relay_number*)
Close relay
Parameters **relay_number** (*int*) – The number of the relay to close

open_relay(*relay_number*)
Open relay
Parameters **relay_number** (*int*) – The number of the relay to open

2.2 Simulator Module

2.2.1 Start_simulator function

simulator.**start_simulator**(*sim_type*, *sim_id*)
This will start a new simulator daemon in a separate process. It has to run the soap daemon with python 2.7, because the used SOAP module is only available for python 2.7.

Parameters

- **sim_type** (*string*) – The type of simulator to start (eg. tcp, udp, etc)
- **sim_id** (*int*) – The id to give to the simulator.

2.2.2 Connection Factory

class connectionfactory.**ConnectionFactory**
This connection factory will return the Dispatcher and Daemon of the requested type. This is the class to use if you want to create and add a new connection to the testsystem.

2.2.3 Daemon

class daemon.**HttpDaemon**(*daemon*, *name*, *daemon_id*)
Class to turn Http simulator into a server that runs in the background.

run()
Start the dispatcher

class daemon.**SerialDaemon**(*daemon*, *name*, *daemon_id*)
Class to turn Serial simulator into a server that runs in the background.

run()
Start the dispatcher

class daemon.**TCPDaemon**(*daemon*, *name*, *daemon_id*)
Class to turn TCP simulator into a server that runs in the background.

run()
Start the dispatcher

class daemon.**TISDaemon**(*dispatcher*, *name*, *daemon_id*)
Baseclass for all TIS daemons Contains the default parameter and logging handling.

setup_logging()
Setup the default logging based on the name of this daemon

startup()
Default parameter handling to start or stop the daemon

class daemon.**UDPDaemon**(*daemon*, *name*, *daemon_id*)
Class to turn UDP simulator into a server that runs in the background.

run()
Start the dispatcher

2.2.4 Dispatcher

class dispatcher.**Dispatcher**(*dispatcher_type*, *dispatcher_id*)
Superclass for all Dispatchers. This is the part of the simulator that handles the connections.

accept(*a_socket*)
Accept a connection from the system.

control_c_handler()
Controlled shutdown so we can cleanup.

process_message(*a_socket*)
Receive and forward a message from the system

process_player_command(*a_socket*)
Process a command from the scenario player.

class dispatcher.**HttpDispatcher**(*dispatcher_type*, *dispatcher_id*)
Dispatcher subclass for Http connections

create_sockets()
Create the UDP sockets between the system and the Scenario player

process_messsage(*a_socket*)
Method to process a HTTP request from the system.

Parameters **a_socket** (*socket*) – the socket on which the message arrives.

class dispatcher.**SerialDispatcher**(*dispatcher_type*, *dispatcher_id*)
Dispatcher subclass for Serial connections

create_sockets()
Create the socket to the scenario player and set up the serial link to the system

process_message()
Receive and forward a message from the system.

process_player_command(*a_socket*)
Process a command from the scenario player.

read_message(*link*)
Read one or more bytes from the system

class dispatcher.**TCPDispatcher**(*dispatcher_type*, *dispatcher_id*)
Dispatcher subclass for TCP connections

create_sockets()
Create the TCP sockets between the system and the Scenario player

class dispatcher.**UDPDispatcher**(*dispatcher_type*, *dispatcher_id*)
Dispatcher subclass for UDP connections

create_sockets()
Create the UDP sockets between the system and the Scenario player

2.3 TestSystem Module

class testsystem.**TestSystem**(*test_system_name*)
Starting point when creating a test system for a specific system.

Use this class by creating a subclass, so you can also add your own simulator interfaces.

add_driver(*name*, *driver*, *driver_id*)
Add a driver to the test system. The configuration of the driver is done with a configuration file. It wil add a driver to the list with the name as key and the specific driver as the value.

Parameters

- **name** (*string*) – The driver name.
- **driver** (*string*) – The kind of driver.

add_scenario_player()
Set the scenario player for the testsystem

add_simulator_interface(*name*, *sim_id*, *sim_interface*)
Add a new simulator interface to the list of simulator_interfaces and also add a socket to the scenario player.

Parameters

- **name** (*string*) – The simulator name.
- **sim_id** (*int*) – The id for the simulator interface.
- **sim_interface** (*string*) – The kind of simulator interface.

add_step(*when*, *step*)
See ScenarioPlayer.add_step()

context = None
This will be the zmq context.

drivers = {}
The drivers used by the testsystem.

kill_previous_script (*test_system_name*)
Kill an already running test script. Can't have to scripts running at the same time.

logger = None
The logger.

play()
See `ScenarioPlayer.play()`

scenario_player = None
The scenario player which will play the steps.

simulator_interfaces = {}
The simulator_interfaces used by this test system.

stop()
See `ScenarioPlayer.stop()`

2.3.1 Testsystem Class

class `testsystem.`**TestSystem**(*test_system_name*)
Starting point when creating a test system for a specific system.

Use this class by creating a subclass, so you can also add your own simulator interfaces.

add_driver(*name*, *driver*, *driver_id*)
Add a driver to the test system. The configuration of the driver is done with a configuration file. It wil add a driver to the list with the name as key and the specific driver as the value.

Parameters

- **name** (*string*) – The driver name.
- **driver** (*string*) – The kind of driver.

add_scenario_player()
Set the scenario player for the testsystem

add_simulator_interface(*name*, *sim_id*, *sim_interface*)
Add a new simulator interface to the list of simulator_interfaces and also add a socket to the scenario player.

Parameters

- **name** (*string*) – The simulator name.
- **sim_id** (*int*) – The id for the simulator interface.
- **sim_interface** (*string*) – The kind of simulator interface.

add_step(*when*, *step*)
See `ScenarioPlayer.add_step()`

context = None
This will be the zmq context.

drivers = {}
The drivers used by the testsystem.

kill_previous_script(*test_system_name*)
Kill an already running test script. Can't have to scripts running at the same time.

logger = None
The logger.

play()
See `ScenarioPlayer.play()`

scenario_player = None
The scenario player which will play the steps.

simulator_interfaces = {}
The simulator_interfaces used by this test system.

stop()
See `ScenarioPlayer.stop()`

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

c

d

s

t

U