# Automated Testing at the GUI level

## Hands-on do it yourself manual

**2023**

**The TESTAR team**

Last updated: 23/05/2023 at 19:09.

**TESTAR research is funded by the following projects.**


ITEA3 TESTOMAT Project will support software teams to strike the right balance by increasing the development speed without sacrificing quality. The project will ultimately result in a Test Automation Improvement Model, which will define key improvement areas in test automation.

`https://www.testomatproject.eu/`


DECODER builds an Integrated Development Environment that combines information from different sources through formal and semi-formal models to deliver software project intelligence to shorten the learning curve of software programmers and maintainers and increase their productivity.

`https://www.decoder-project.eu/`


Extended Reality (XR) systems are advanced interactive systems such as Virtual Reality (VR) and Augmented Reality (AR) systems. IV4XR aims to build a novel verification and validation technology for XR systems with AI techniques to provide learning and reasoning in a virtual world.

`https://iv4xr-project.eu/`


IVVES will systematically develop Artificial Intelligence approaches for robust and comprehensive, industrial-grade V&V of "embedded AI", i.e. machine-learning for control of complex, mission-critical evolving systems and services covering the major industrial domains in Europe.

`https://ivves.weebly.com/`

ENACTEST project works on identifying and designing early and seamless teaching materials for testing that are aligned with industry needs and which take into account also the learning needs and characteristics of students.

`https://enactest-project.eu/`

# Contents

# About this Hands-on

## For whom is this document?

This manual is meant for students and interested parties, to be used as an introduction to the TESTAR tool for automated testing through the Graphical User Interface (GUI).

## How to read this document

This document contains tasks to familiarize oneself with TESTAR's workings. A task is marked like this:

**hands-on 0**

A task to perform.

Instructions on how to perform the required action(s).

# Glossary

**API** Application Programming Interface: a set of rules and specifications with which software can communicate with other software to access and make use of services and resources. 50, 93

**Graphical User Interface (GUI)** The graphical part of a program. 6, 8, 14, 16, 17, 19, 21, 22, 25, 26, 28, 34, 37, 38, 59, 60, 70, 74, 79, 82

**GUI testing** A testing technique where one tests the SUT solely through its GUI. 8

**oracle** The mechanism that determines whether a test has succeeded or failed. 18, 21, 22

**Regular Expression (RegEx)** . 32

**sequence** A series of actions for the purpose of testing. 17, 22

**System Under Test (SUT)** The application that is being tested. 8, 14, 16, 17, 19, 21–25, 27, 31, 33, 35, 43–45, 50, 53, 57, 70, 74, 81, 82, 93, 102–104

**User Interface (UI)** . 22, 95, 100–104

**Virtual Machine (VM)** The aspects of a program that allows humans to interact with it. 9, 10, 15, 22, 34, 93, 96

**widget** An element of interaction in a GUI, such as a button, a text field, or a scroll bar. 19

SECTION 1

# What is TESTAR

TESTAR[1] is an open source[2] tool that implements a scriptless approach for completely automated test generation at the Graphical User Interface (GUI) level for Web and Windows desktop applications.

TESTAR is based on agents that implement various action selection mechanisms and test oracles. The underlying principles are very simple: generate test sequences of (state, action)-pairs by starting up the System Under Test (SUT) in its initial state and continuously selecting an action to bring the SUT into another state that is checked by oracles for failures.

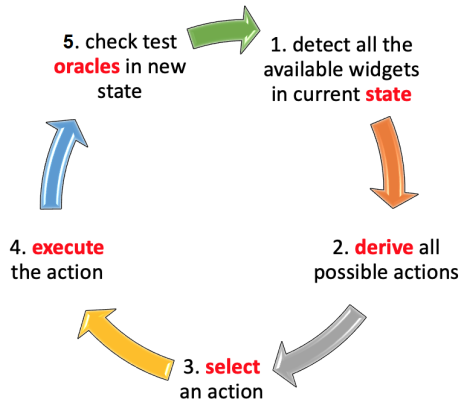



Figure 1: TESTAR underlying principles

The action selection characterizes the fundamental challenge of intelligent systems: what to do next. The difficult part is optimizing the action selection to find faults and recognizing a faulty state when it is found. Faulty states are not restricted to functionality errors, but also violations of other quality characteristics, like accessibility or security, can be detected by inspecting the state.

TESTAR shifts the paradigm of GUI testing: from developing scripts to developing intelligent AI-enabled agents.

[1] https://testar.org/
[2] https://github.com/TESTARtool/TESTAR_dev

SECTION 2

# Preparation and installation of TESTAR

To be on the safe side, you should run TESTAR within a Virtual Machine (VM). TESTAR can do random things, and this way, you make sure you do not break anything in your operating system. For example, when testing an application like Notepad, TESTAR can randomly interact with the system files through Notepad Windows Explorer features. To configure TESTAR to avoid these random things, the user needs to pay attention to this HandsOn guide ;)

There are two options to try out TESTAR in a VM:

- Run TESTAR in a remote VM that runs on one of our servers. You can access the VM through the browser (you need to request credentials, see 2.1 "Running TESTAR on a server" on this page)
- Download an (OVA) image of a VM and install it on your local machine (see 2.2 "Download and install a virtual machine image" on the current page and 2.3 "Installing TESTAR" on the following page)

If you have your own Virtual Machine, then you can download and execute the TESTAR binaries directly (see 2.3 "Installing TESTAR" on the next page)

## 2.1 Running TESTAR on a server

We use Apache Guacamole[3], a clientless remote desktop gateway, to enable a remote connection to a VM desktop using a web browser. This is called clientless because no plugins or client software are required.

For this option, you have to request a server address and credentials. This will allow you to connect to a TESTAR VM with all the required software already installed using your web browser. This can be done by sending an email to info@testar.org with the subject "TESTAR remote VM connection request".

To log in, go to `https://qdesktop.testar.org/` and enter your assigned username and password. Once you are connected to the virtual machine, you can enable the clipboard-sharing permissions in the search bar of the browser.

## 2.2 Download and install a virtual machine image

A Virtual Machine (OVA) image is available with all the required software already installed. To run the image, you need to have VirtualBox version 5.2.18 or later. If you do not have VirtualBox 5.2.18 on your machine, you can download it here:
`https://download.virtualbox.org/virtualbox/5.2.18/`

If you have a newer version (> 5.2.18) of VirtualBox installed, you might need to update

[3] `https://guacamole.apache.org/`

the “VBox Guest Additions” on the image to the same version as your VirtualBox version. Please refer to the VirtualBox manual on how to do this.

Minimal resource requirements for the VirtualBox host:

- VirtualBox version $\geq$ 5.2.18
- Recent Dual core CPU or better
- 8 GB Memory
- about 80 GB of free disk space

The hands-on OVA image can be downloaded from the TESTAR website:

`https://testar.org/images/`<*image_name*>

You will be given <*image_name*> during the training-sessions. If you are doing the hands-on on our own, please send an email to `info@testar.org` to obtain the latest image or check from `https://testar.org/download/` and look for the latest OVA image.

**NOTE**: The OVA image is about 22GB downloaded and takes up about 50GB once the virtual machine is created. It will grow in size with usage and is maximized at 200GB, but with normal usage it should not grow beyond 60GB during this hands-on.

**hands-on 1**

Set up and run the Virtual Machine in VirtualBox. Log in as the “testar” user.

Import the downloaded OVA image in VirtualBox via “File” → “Import Appliance”. Use the account below to log in on the VM:
username: ”`testar`”
password: ”`testar`”

## 2.3 Installing TESTAR

To run TESTAR locally on your own computer, you can download the testar_<version number>.zip from the TESTAR website and unpack the zip file into the `C:\` directory with the instructions below.

**hands-on 2**

On your own VM, download and unpack the TESTAR zip.

- Go to `https://testar.org/download/`
- Download the latest TESTAR zip distribution “TESTAR master latest binaries”.
- Go to the windows downloads directory and right-click on the just downloaded testar_<version number>.zip file. Select the “Extract ALL” option in the menu. (see Figure 2)
- Select for the destination directory `C:\` . Click the “Extract” button, this will extract the testar_<version number>.zip file, the installation can be

found in `C:\testar\` (see Figure 3)

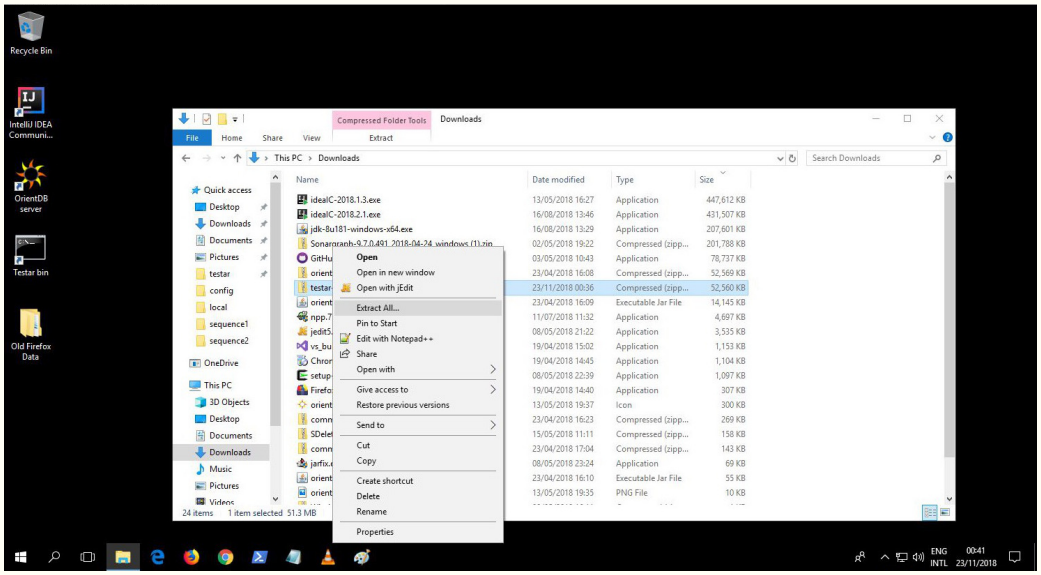



Figure 2: Right click and select “Extract all”

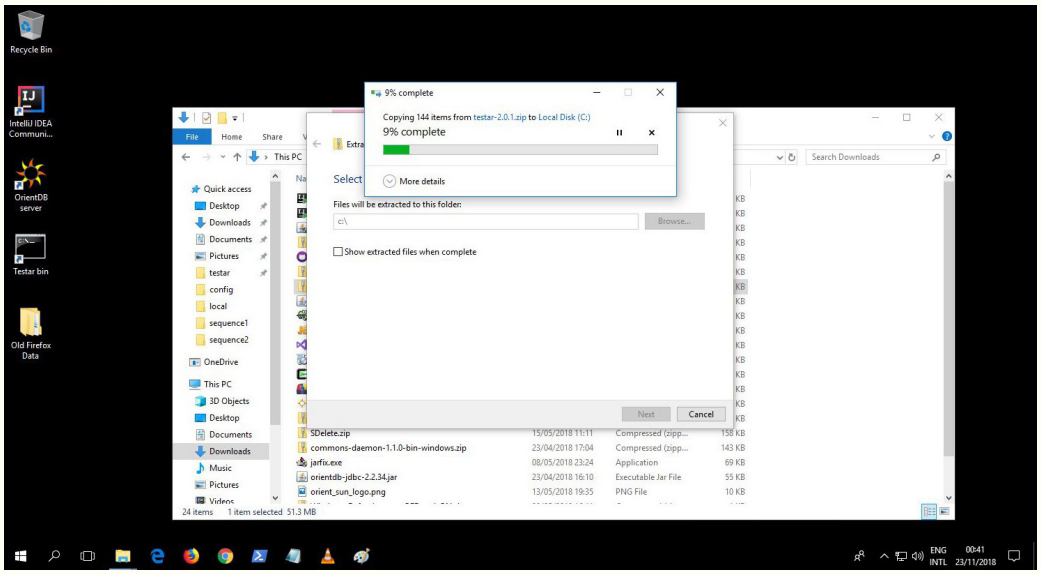



Figure 3: Unzip testar_<version number>.zip file in windows

## 2.4 Java versions

We are currently maintaining TESTAR to support three main Java versions:

1. Java 8 is implemented into two products: JDK 8 (Java SE Development Kit 8) and JRE 8 (Java SE Runtime Environment 8). The runtime execution and compilation of TESTAR protocols require the installation of `Java 8 JDK`

2. Java 11 JDK contains all the TESTAR requirements. This is currently the recommended version to follow these Hands-On exercises.
3. Java 17 JDK contains almost all the TESTAR requirements. However, this version does not include the Nashor Script Engine feature in the Java package, which provokes the use of the Edit Protocol feature without highlighting code with colors (but it is still a functional feature).

## 2.5 Quick tips

Before starting, we want to show you a trick to open a command prompt in the selected Windows directory. Figure 4 indicates how typing "cmd" or "cmd.exe" in the explorer bar allows users to open the command prompt in the `testar\bin\` directory quickly.

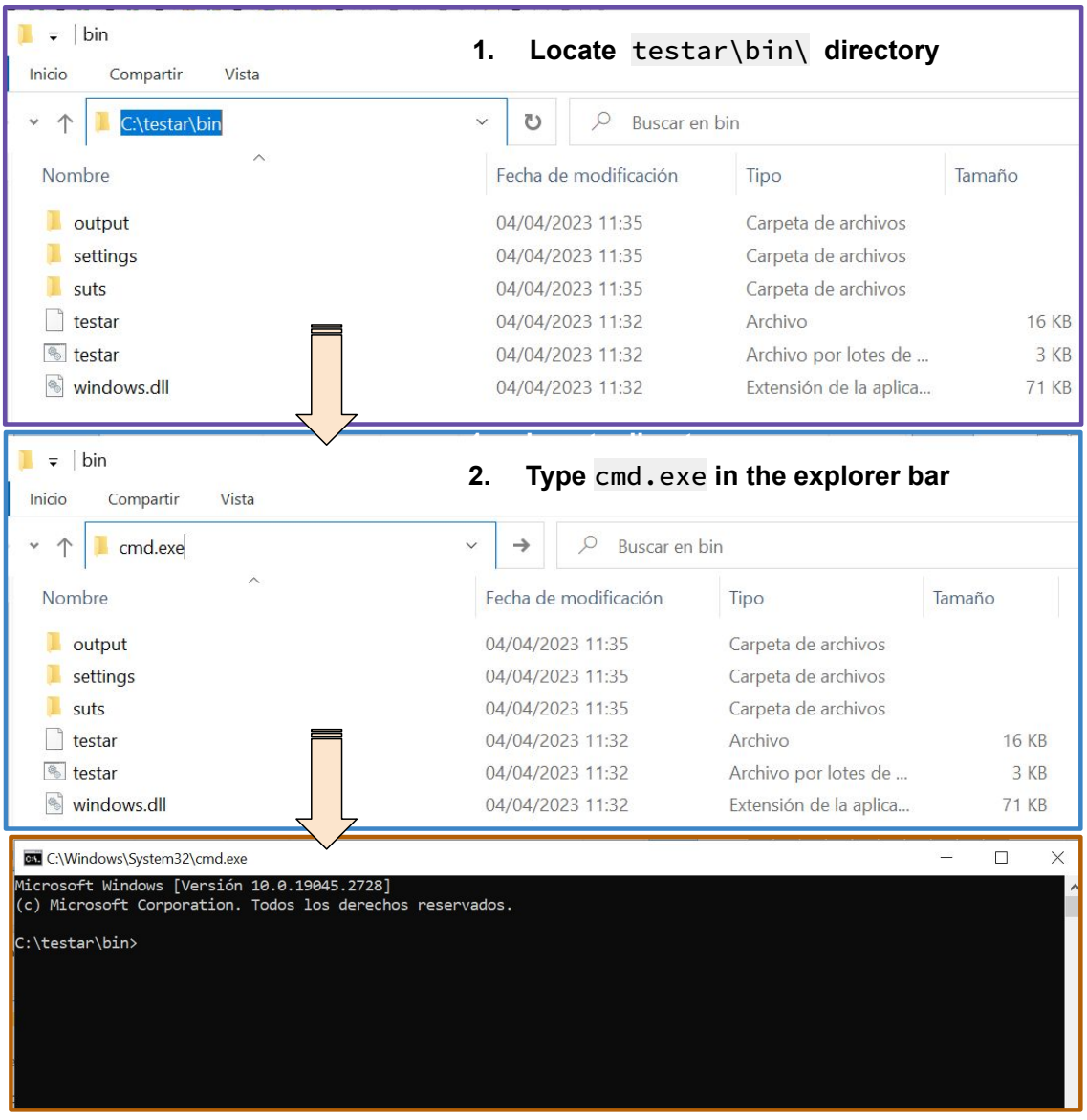



Figure 4: How to open a command prompt in a Windows directory

Conversely, Figure 5 indicates how typing “start .” in a command prompt allows users to open a Windows Explorer directory.

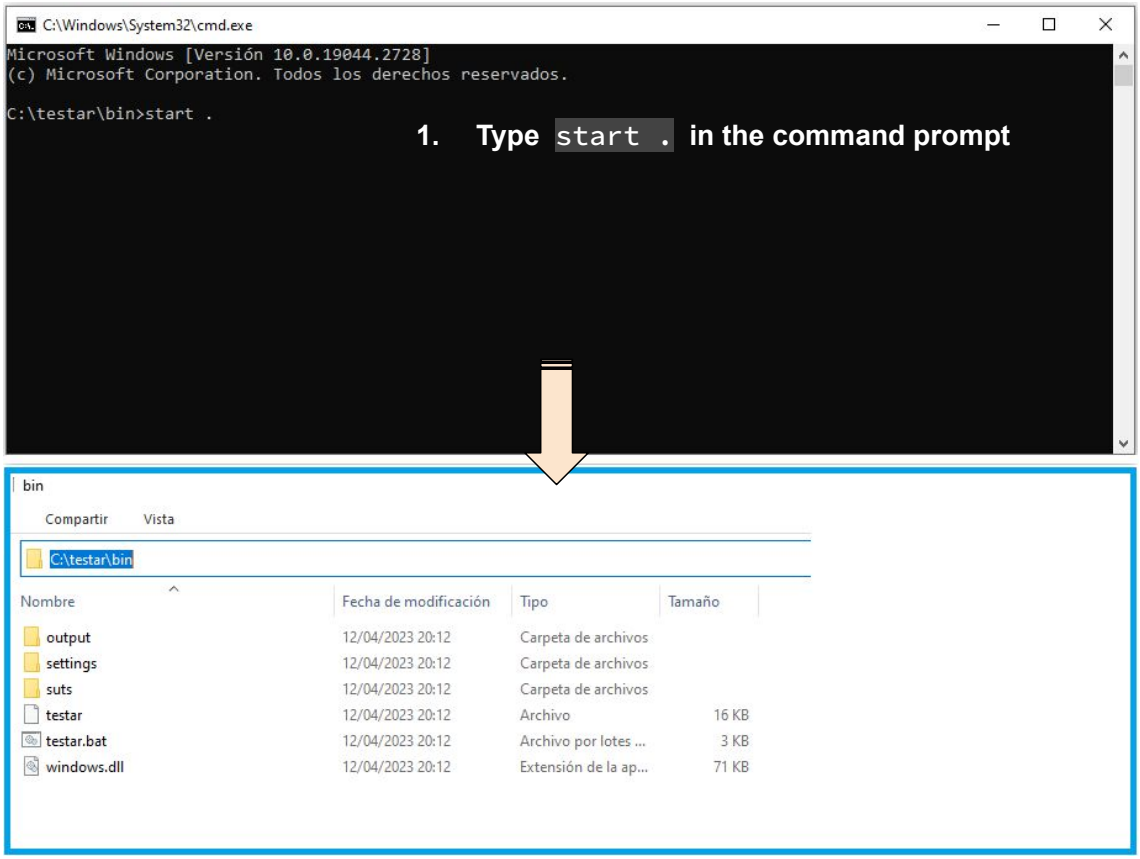



Figure 5: How to open a Windows Explorer directory from a command prompt

SECTION 3

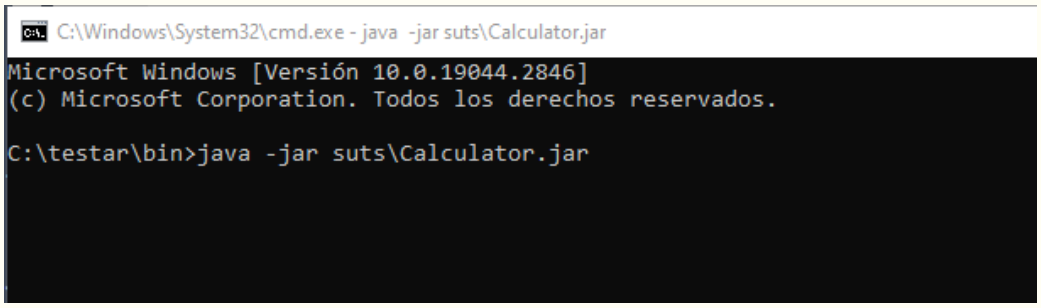
# Testing at the GUI level

**hands-on 3**

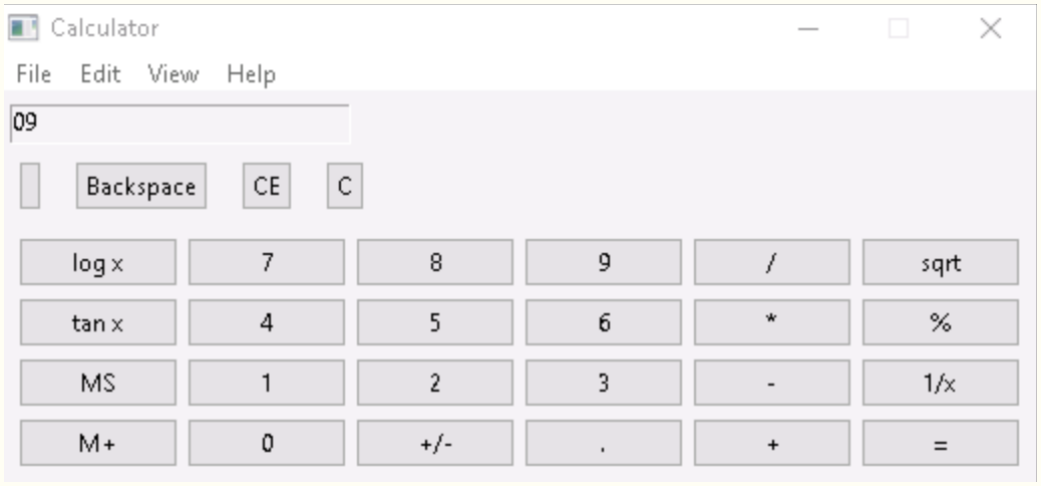
As a vehicle to explain different parts of TESTAR, we will use a simple Calculator as SUT. We have prepared the Calculator Java application with several faults. To get a first impression of this SUT, this task will ask you to test the Calculator program manually at the GUI level. How many and what type of failures can you find?

The Calculator is found in the `testar\bin\suts\Calculator.jar` folder as part of the TESTAR binaries. To start up the Calculator, open a command prompt in the `testar\bin\suts` directory and type:

`java -jar suts\Calculator.jar`.




You will see a desktop application looking like this:




***TS Note:*** If you found some Java troubleshooting you can find help in the APPENDIX section A.

SECTION 4

# Getting TESTAR running

## 4.1 Starting up TESTAR

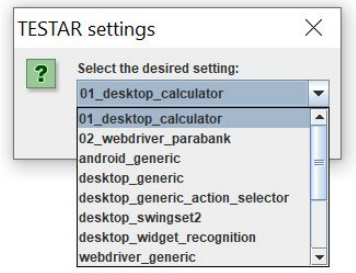
In the `bin` directory of the unpacked `testar`-directory, there is a `testar.bat`. If you are using the Virtual Machine, you can use the shortcut on the desktop instead.

**hands-on 4**

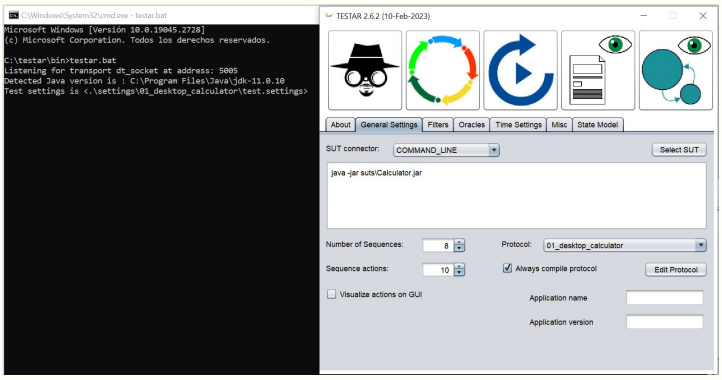
Start TESTAR by running testar.bat

Run a command prompt at the location `testar/bin/`, type in `testar.bat`, and press Enter .

The first time TESTAR is executed, the tool does not have any selected protocol configuration. When this happens, TESTAR opens a dropdown box for the user to select a protocol. You can select the `01_desktop_calculator` protocol:




After selecting, the TESTAR Dialogue will appear like this:




Try not to click anywhere on the TESTAR Dialogue yet. First, you should read the next section which gives a short introduction about the impressions that TESTAR Settings Dialog offers to the users.

***TS Note:*** If everything worked as described above and the TESTAR Dialogue has started up, then continue with section 5. If not, some troubleshooting help can be found in the APPENDIX section A.

SECTION 5

# Introduction to TESTAR's Settings Dialogue

Now that we have the TESTAR Dialogue running, in the following sections, we are going to start from the surface and continue to dive deeper into the mysteries of TESTAR, on the journey to becoming an advanced user.

The Settings Dialogue is a representation of the settings and configuration of TESTAR. It provides a visual way for configuring the values that are present in the test.settings file. These settings define details for TESTAR on how to test a specific SUT.

As you can see in Figure 6, we are going to start by introducing the five different "sections" of the General functionality (colored accordingly in the Figure below).

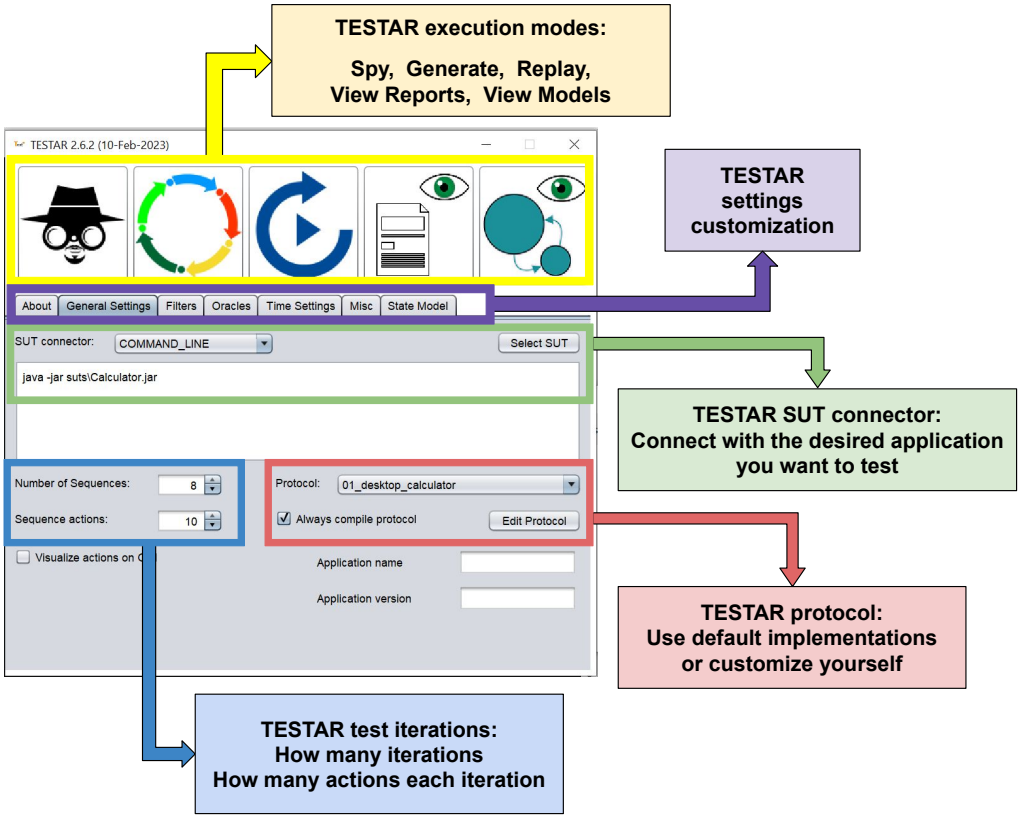



Figure 6: TESTAR's Settings Dialogue

**TESTAR execution modes (Yellow)** :

These icons represent the different modes that you can use to inspect or test the desired application/SUT. In order, these are:

SPY (Check the SUT widgets that TESTAR can recognise at the GUI level
GENERATE (Generate test sequences automatically)
REPLAY (Re-execute an existing test sequence)
VIEW_REPORTS (See an action-by-action HTML sequence report)
VIEW_MODELS (See the inferred TESTAR State Models)

**TESTAR settings customisation (Purple)** :

Consists of tabs that we can use to navigate and customise different settings, like, for example:

- General Settings
- Filters
- Oracles
- Time Settings
- etc.

In Figure 6 the "General Settings"-tab is selected, and there we can see, for example, the following options that can be configured:

**TESTAR SUT-connector (Green)** :

Here we will define which application or System Under Test (SUT) we want to test and how to connect with it.
In Figure 6, we can see the default is to connect through `COMMAND_LINE`.
The SUT that is selected can be started up through the command line by the following command: `java -jar suts/Calculator.jar`
This indicates to TESTAR that you want to launch and connect to the executable of a Java Calculator.
More details on connecting to SUT can be found in Chapter 7.

**TESTAR tests iterations (Blue)** :

These are settings for GENERATE mode to choose how many sequences you want to generate and how many actions each sequence will contain.

**TESTAR protocol (Red)** :

A TESTAR protocol is a Java class that is responsible for executing the different parts of TESTAR's workflow for generating test-sequences that we saw before:

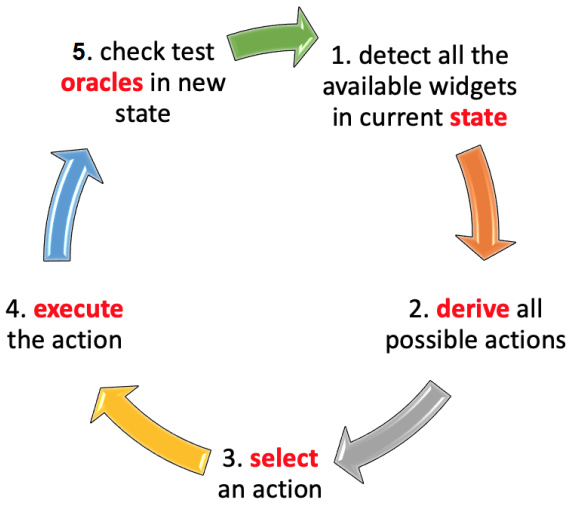



Figure 7: TESTAR's workflow

- (1) Obtaining the GUI state (`get_state`).
- (2) Deriving the set of actions that a potential user can execute in that specific state (`derive_actions`).
- (3) Selecting one of these actions (`select_action`).
- (4) Executing the action (`execute_action`).

- (5) Evaluating the new state using existing oracles to find failures (`get_verdict`).

The "Always compile protocol" checkbox indicates whether the protocol must be compiled at run-time before starting a TESTAR execution mode. It can be modified by accessing the Java sources in an editor that opens through the "Edit Protocol" button in the Dialogue. You can, of course, use your own Java IDE to edit the source code. The source code allows you to write much more fine-grained implementations of the methods implementing the workflow.
From the drop-down menu "Protocol", we can select the protocol we want to use for testing. In Figure 6, we selected *01_desktop_calculator* protocol. Other protocols like *02_webdriver_parabank* for web applications (see section 8) or *android_generic* for Android systems (see section 11) will be used later.

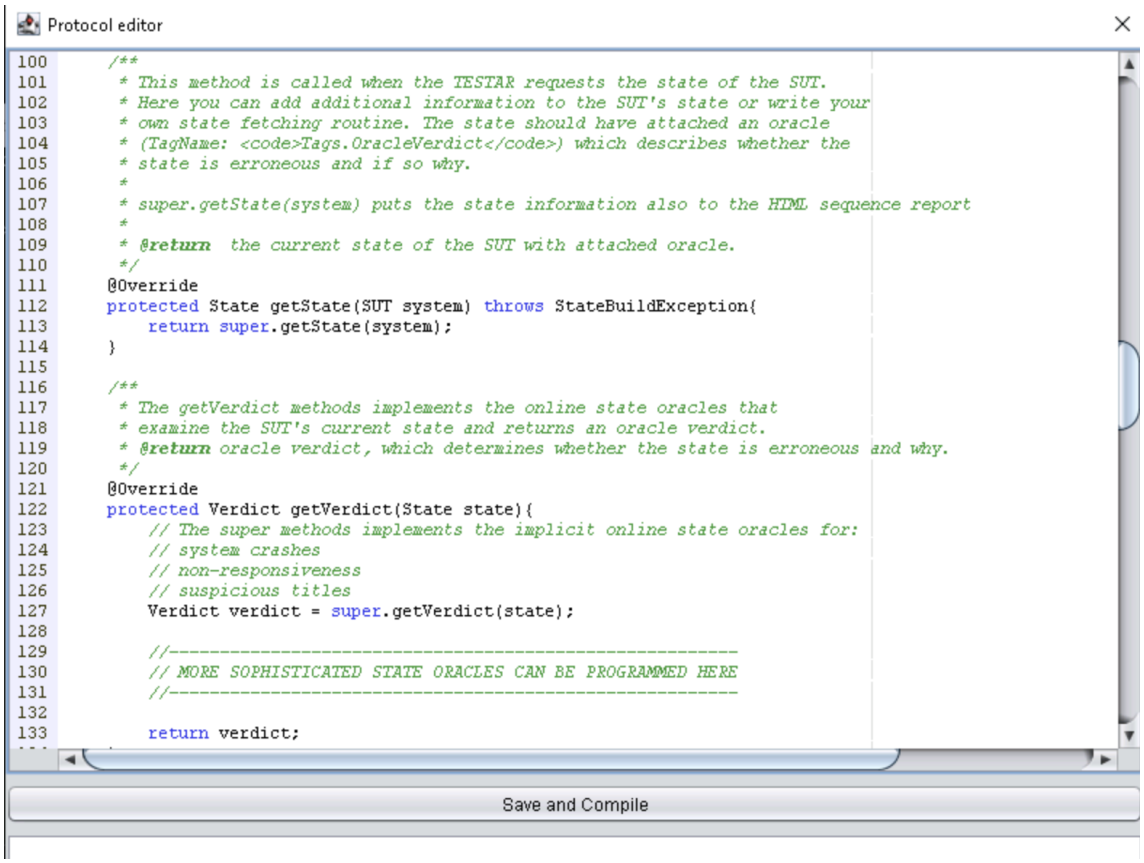



Figure 8: The protocol editor

**hands-on 5**

Explore the TESTAR protocol

With the *01_desktop_calculator* protocol selected, click the "Edit Protocol" button in the in the "General Settings"-Tab of the TESTAR Dialog. The protocol editor will open. You will see something like this in Figure 8.

Browse through the code and try to find the methods that are implementing TESTAR's workflow from Figure 7:

- `get_state`
- `derive_actions`
- `select_action`
- `execute_action`
- `get_verdict`

If you do not understand the code, do not worry for now. The objective is now to recognize where the main workflow of TESTAR is implemented and could potentially be changed by you as the tester.

In the next section, we will start looking at the SPY mode: it enables us to spy the buttons and other widgets of the SUT and see all the information that TESTAR is able to extract.

## 5.1 SPY Mode

**hands-on 6**

Inspect a SUT with the SPY mode

Now, within the *01_desktop_calculator* protocol, click on the SPY button on the top left side of the TESTAR Dialogue (the one with the magnifying class). Hover over the different parts of the GUI and look for yourself.

***TS Note:*** If you have not been able to correctly detect the widgets with the mouse (by moving it to widgets screen coordinates), check the instructions in Appendix B.

The SPY mode helps you to configure TESTAR by allowing you to inspect the widget controls of the GUI. In spy mode, you can:

- See what actions TESTAR is able to derive and choose from. The green dots (like in Figure 9) represent the available widgets a user can click on in that specific state of the GUI.
- Hover over an element to show the properties and information of that element or widget, like in Figure 10. `NOTE:` Maximise button does not have a green dot because TESTAR detects that the button is not Enabled (see Figure 11).
- Press Shift + ↑ to show extended information and properties while hovering over an element. This way, you can find more details, for example, about the titles of the elements. To go back to less information, just press Shift + ↑ again.

To leave SPY mode and return to the TESTAR start-up dialog, press Shift + ↓, or close the system you were spying.


Figure 9: Calculator in SPY mode

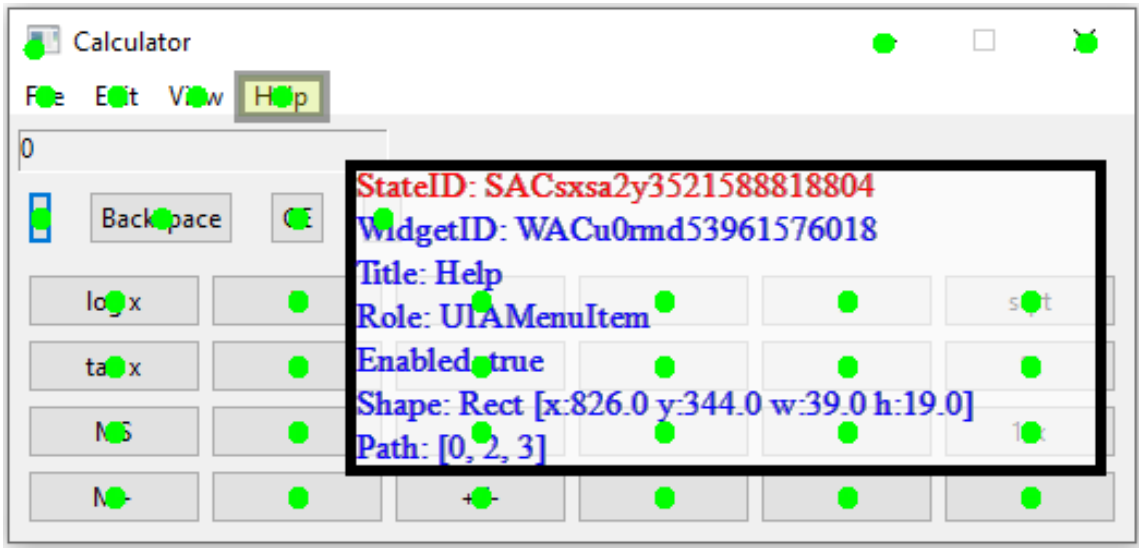



Figure 10: Calculator in SPY mode hover over elements

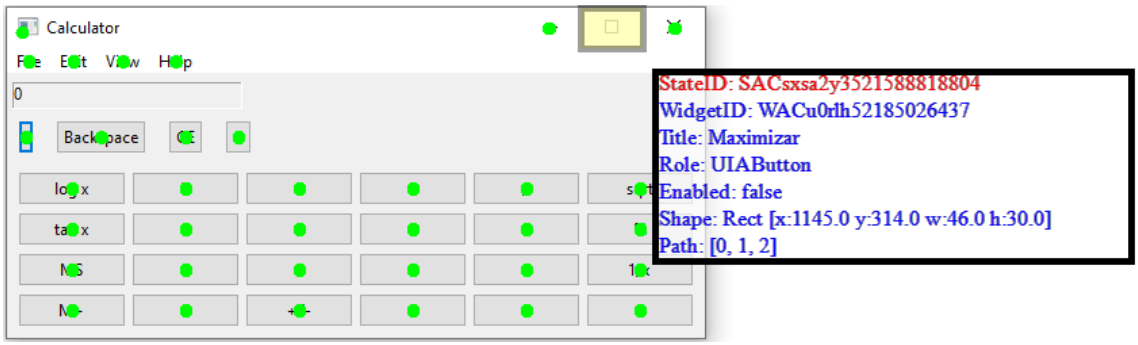



Figure 11: Calculator in SPY mode shows that Maximise is not Enabled

## 5.2 GENERATE mode

In this mode, the TESTAR tool carries out automated testing following the test workflow we saw already in Figures 1 and 7. In Figure 12 you can find an extended version of this workflow.

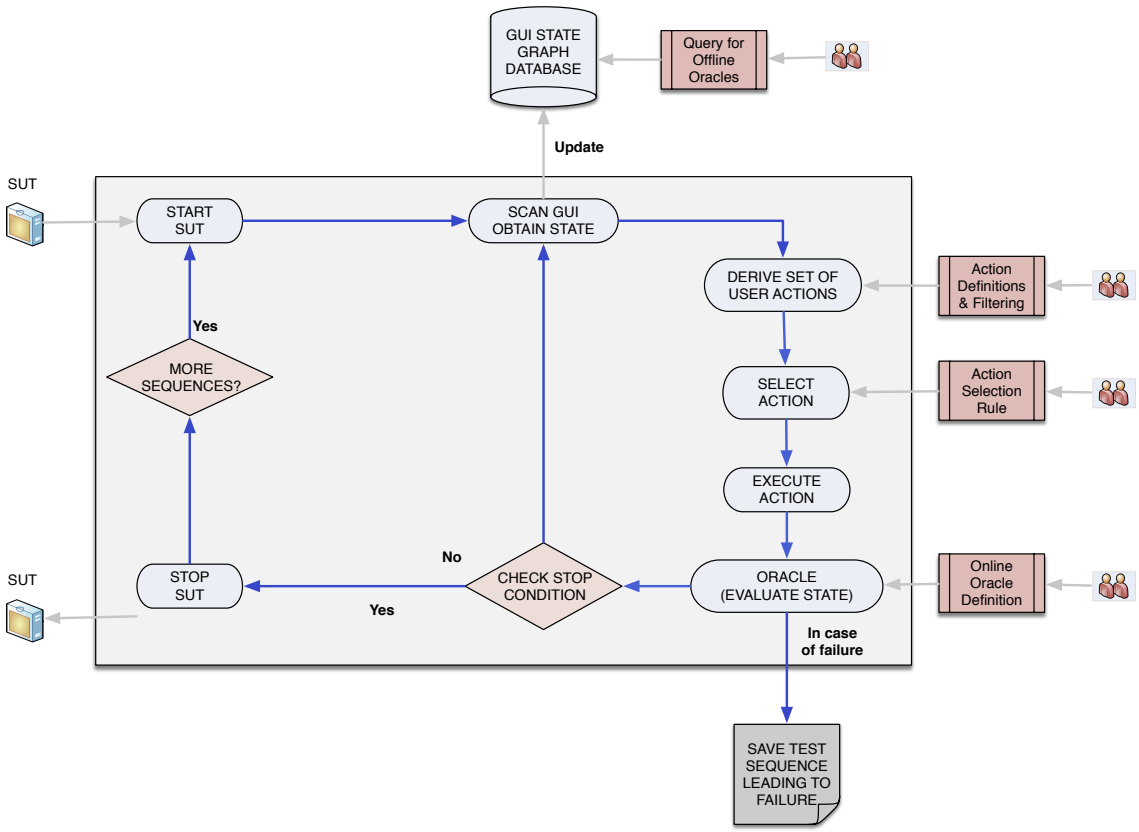



Figure 12: TESTAR test flow

Again, you can see, after starting the SUT, recognise the workflow:

- Obtaining the GUI state (`get_state`).
- Deriving the set of actions that a potential user can execute in that specific state (`derive_actions`).
- Selecting one of these actions (`select_action`).
- Executing the actions (`execute_action`).
- Evaluating the new state using the oracles to find failures (`get_verdict`).

Basically, TESTAR derives a set of possible actions for the current state that the GUI of the SUT is in, such as clicking on the elements that we have visualized with the green dots. Then, it automatically selects and executes an action from this set which makes the SUT go to a new GUI state. This new state is evaluated with the available oracles. If no failure is found, again, a set of possible actions for the new state is derived, one action is selected and executed, etc. This loop continues until a failure is found or a stopping criterion is reached. With the proper test setup, all you will need to do is to wait for your tests to finish.

The default behavior includes a random selection of actions and implicit oracles for the

detection of the violation of general-purpose system requirements:

- the SUT should not crash,
- the SUT should not find itself in an unresponsive state (freeze), and
- the User Interface (UI) state should not contain any widget with suspicious titles like *error*, *problem*, *exception*, etc.

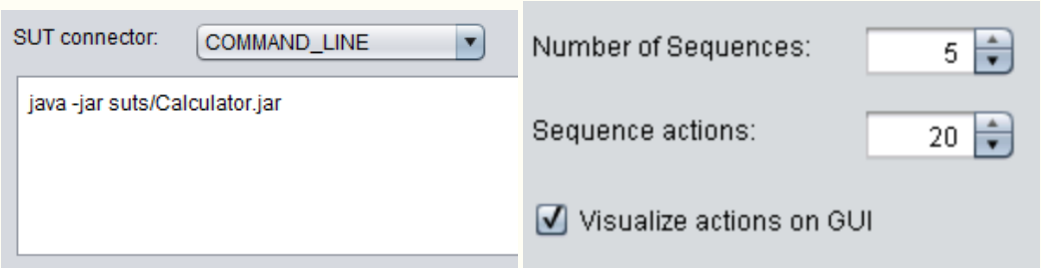
Now we are ready to do some automated testing with TESTAR. With random action selection, these are known as monkey tests. TESTAR can *see* the controls of the SUT's UI and automatically detect possible actions. It randomly selects and executes these actions.

**hands-on 7**

Trying out the TESTAR tool as a "dumb monkey", using purely random action selection strategy and only the default test oracles by generating test sequences for the Calculator SUT.

Let us alter the TESTAR configuration to start this Calculator application in the "General Settings"-Tab:

- Check it is selected, or select the SUT connector `COMMAND_LINE` from the drop-down and type the following SUT in the textbox:
  `java -jar suts/Calculator.jar`
- Configure the number of test sequences you want to generate and the length (i.e., number of actions) of these sequences. Since we are learning the tool, let us not put too many. Select, for example, 5 sequences of 20 actions.
- Turn on the visualization by checking the corresponding checkbox.




Now, click on the GENERATE button (the icon with multicolored arrows) to start running tests as specified by the test setup. See what happens with the Calculator during the test runs. Even the generic test oracles should find some crashes.

The following color codes are applied for visualization during the test:

- **green dots** for UI actions that TESTAR can detect and execute, and
- **gray dots** for UI actions that TESTAR can detect but are filtered, and
- **red dot** for the currently selected UI action being executed.

*Warning:* It is good to realize that random GUI testing may have strange results, for example, if the SUT permits to open, save, or delete system files. Often, it is safer to run random testing on a Virtual Machine that can be easily restored if

something breaks. TESTAR has some safety measures to prevent this, but better be safe than sorry.

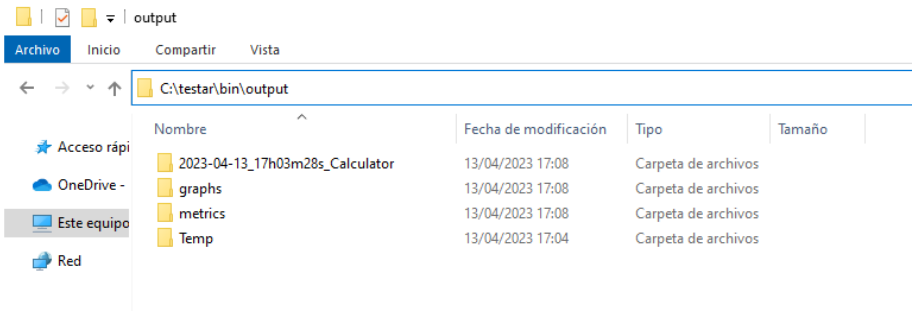
***TS Note:*** If TESTAR does not execute in the correct coordinates of the selected **red dot** action you can find help in the APPENDIX sections B and C

When the tests have finished, we can inspect the results of our tests. This is explained in the next section.

You could also go through the generated HTML reports to see what kind of screenshots were found during the test runs and whether there are screens that look like a failure. You should be able to see dialogs that you would report as bugs.

## 5.3 Test Results and Reports

When a test run has finished, we can inspect the results in the `\testar\bin\output\` directory. It looks, for example, like this:




The names of the folders have the format (timestamp = yyyy-mm-dd_hh-mm-ss)

`timestamp_SUTname`

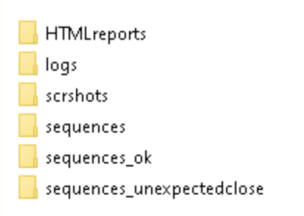
**hands-on 8**

Find the location where the results of a test run are stored.

The results of our tests are stored in the directory:

`\testar\bin\output\`

There you will see the generated folder for the test run that starts with a date, then a time, and then the name of the SUT (i.e. *Calculator*). If you open the folder, you will probably see different folders like:




The `HTMLreports`, `logs`, `scrshots`, and `sequences` folders contain the 5 sequences you just run in different formats that will be explained below.

By default, TESTAR will try to obtain the name of the SUT application (i.e., `SUTname`) by reading the path of the executed application. However, sometimes, when more clear names are desired, it can be better to configure the name in the "Application name" field in the "General Settings"-Tab of the TESTAR Dialogue.

**hands-on 9**

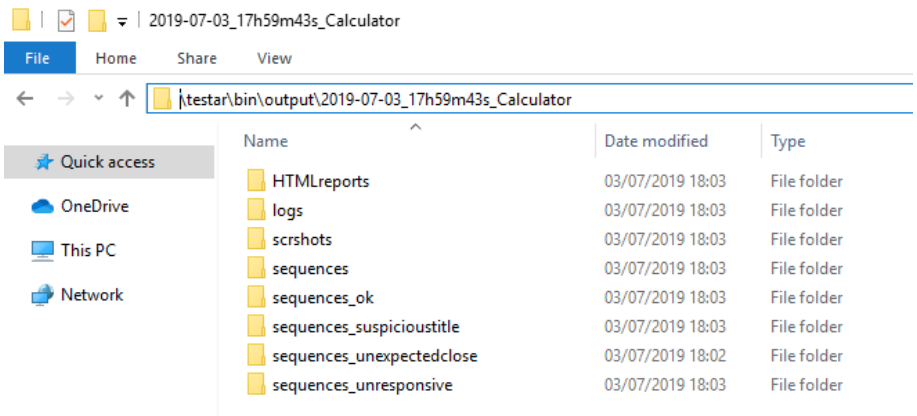
Change the name of the folder containing the test results.

In the Dialogue, go to the "General Settings"-Tab and find the following in the "Application" fields (see also Figure 6).


Put a new name for the Calculator SUT and run some tests again. When the tests have finished, go to the `\testar\bin\output\` directory and see that now a folder has been created with the name you have chosen for the application.

Try the same with the version number.

The folders inside the directory `\testar\bin\output\` directory contain all reports and sequences that TESTAR generated during the test run the folder represents. The reports and sequences results can be divided into up to 8 directories as you can see below for a test-run with the Calculator:



Below we will discuss the contents of each of the directories next, then we will VIEW and REPLAY some of the sequences to actually see the results.

**HTMLreports** This folder contains HTML reports detailing the different states, widgets and actions that were found and executed by TESTAR during the run.

**logs** Details about the executed actions and their attached Widget in plain test format.

**scrshots** This folder contains the screenshots of the GUI state taken by TESTAR throughout the execution of actions.

**sequences** This folder contains all the sequences that have been executed by TESTAR. The files in these directories are binary files (with the extension `.testar`) and can only be read by TESTAR in REPLAY mode. We will look at this mode in the next section.

**sequences_ok** This folder contains all those sequences from the folder **sequences** that have not resulted in failure.

**sequences_suspicioustitle** This folder contains all those sequences from the folder **sequences** that have triggered the suspicious title oracle and resulted in failure.

**sequences_unexepectedclose** This folder contains all those sequences from the folder **sequences** that resulted in a crash.

**sequences_unresponsive** This folder contains all those sequences from the folder **sequences** that resulted in the SUT being unresponsive.

TESTAR also generates a summary log report that contains the timestamp, directory, file and result of all the executed sequences. (**NOTE**: this file will be created from the first run on and extended each run).

```
\testar\bin\index.log
```

And a set of detailed logs to investigate what happened during the execution of a specific sequence. You can find these in:

```
\testar\bin\output\timestamp_SUTname\logs
```

These logs, especially the `index.log`, could be useful to integrate TESTAR tool and the results of the sequences into a CI pipeline.

## 5.4 VIEW Report Mode

The execution of the VIEW mode opens a pop-up that allows you to select the HTML report of the sequence you want to view.

In this mode, you can choose an HTML report of the sequence or a plain text `.log` file. TESTAR will use the default system applications defined as a visualization tool (Microsoft Edge and Notepad into the Virtual Machine) to open the selected files.

For example, in Figure 13, you can see the HTML report of or a test sequence executed with the buggy calculator that contains the obtained information by TESTAR about the founded States, Widgets, and available Actions.

A TESTAR sequence can generate an extensive HTML report due it includes all step-by-step states and derived actions. For this reason, the top-left corner of the HTML report

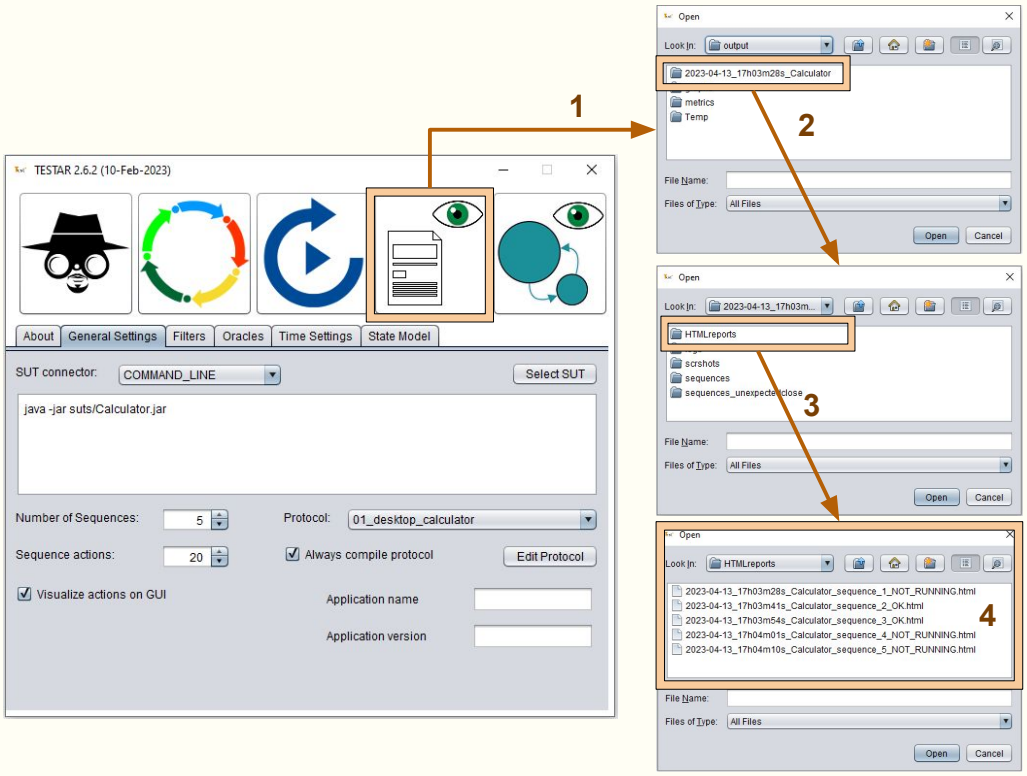
allows users to reverse the order in case they wish to check the final Oracle quickly.


Figure 13: Visualize HTML report

**hands-on 10**

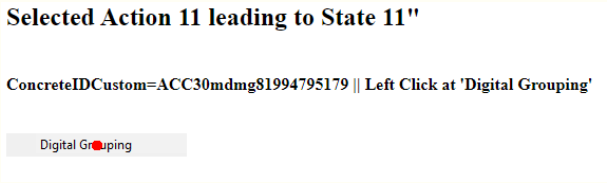
View the HTML results of a test run of the Calculator.

Start up TESTAR and click the VIEW Report button on the TESTAR GUI. Select the HTML one of your previously executed test sequences, and browse through it to observe the information it provides for you.




Now open an HTML related to a sequence that ended with an unexpected close. Why did the SUT close unexpectedly?

One reason can, for example, be the following buggy behavior:




TESTAR selected the action `Left Click at 'Digital Grouping'` which contains an internal failure that crashes the Calculator.

However, another example can be the following false positive behavior:


TESTAR selected the action `Left Click at 'Cerrar'`, meaning it clicked the close button (cerrar in Spanish) and closed the SUT.

This second false positive action is, evidently, undesired and not optimal for the testing process. You can see that TESTAR detects it as a "crash" whenever it closes the main window. Obviously, the tool does not know that closing the main window terminates the application. The human tester does know that. Later on, we can see how we can filter out the selection of these undesired actions.

**hands-on 11**

View the log results of a test run.

Start up TESTAR and click the VIEW Report button on the TESTAR GUI. Browse to the `logs` directory and select the log of one of your previously executed test sequences. Try to understand the information that the file is presenting to you.

## 5.5 REPLAY mode

When you click the REPLAY button, a file explorer window allows you to select the sequence that you want to replay or view.

The REPLAY mode can replay sequence files that end with the extension `.testar`. These files.testar are those that can be found in the directories of the different sequences.

REPLAY will play the saved sequence again, repeating the same actions executed previously. This can be used to verify that a correct sequence (`sequences_ok`) still running without failure after the release of a new version, or to check that the found failures are solved with a fix release.

**hands-on 12**

Replay the sequence of a test run.

Start up TESTAR and click on the REPLAY button on the top right side of the TESTAR GUI. Select a .testar sequence of your previously executed test sequences and replay it.

`NOTE:` If you select a failed sequence to be replayed, TESTAR will probably find the same failure again.

## 5.6 The test.settings file

As indicated before, the start-up Dialogue of TESTAR enables us to easily configure the values that are present in the test.settings file. These files, however, can also be edited directly with a text editor. You can find the test.settings files under the *settings* folder in TESTAR's bin folder (`\testar\bin\settings`) Each test settings configuration is stored inside a unique subfolder (e.g., *01_desktop_calculator*), which contains:

1. a Java source file (e.g., `Protocol_01_desktop_calculator.java`) with the programmable test protocol that implements the workflow.
2. a test.settings file, which contains a list of test properties.
3. you might also see a `protocol_filter.xml` file (this will be discussed later).

`IMPORTANT:` The TESTAR dialog accesses and edits both test.settings and Java protocol files. If you have the dialog opened and try to change something in the test.settings or Java protocol using, for example, Notepad++, these changes will not be saved. Therefore, before manually editing the test.settings or Java protocol files, it is better to close the TESTAR dialog.

In the settings directory we can see a file ending with the extension `.sse`. This file is used to indicate from which folder TESTAR will choose the settings. This protocol selection can be changed by editing the file.sse name directly or through the TESTAR user interface, selecting the desired protocol from the corresponding dropdown-menu in the "General Settings"-Tab.

If the `.sse` is not present in the *settings* folder, a dialogue like the one in Figure 14 will start up to ask you to select the protocol that you would like.

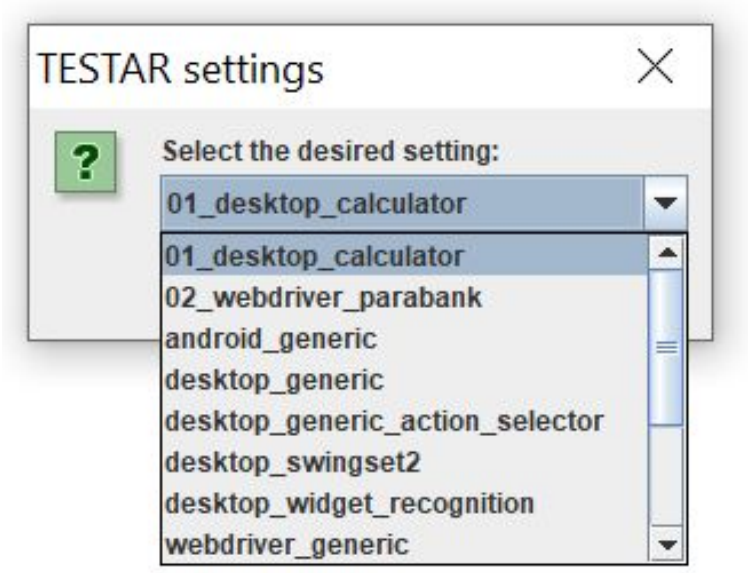



Figure 14: TESTAR when started without an .sse file

If you want, you can also edit the files directly. For now, to illustrate, we refer to Figure 15, where you can see part of the test.settings file for the calculator application. More settings are explained in detail in Appendix J.

```
######################################################################
# TESTAR mode
#
# Set the mode you want TESTAR to start in: Spy, Generate, Replay
######################################################################

Mode = Spy

######################################################################
# Connect to the System Under Test (SUT)
#
# SUTCONNECTOR = COMMAND_LINE, SUTConnectorValue property must
# be a commandline that starts the SUT. It should work when
# typed into a command prompt (e.g. java -jar SUTs/calc.jar). For
# web applications, follow this format: web_browser_path SUT_URL.
#
# SUTCONNECTOR = SUT_WINDOW_TITLE, then SUTConnectorValue property
# must be the title displayed in the SUT' main window. The SUT
# must be manually started and closed.
#
# SUTCONNECTOR = SUT_PROCESS_NAME: SUTConnectorValue property must
# be the process name of the SUT. The SUT must be manually started
# and closed.
######################################################################

SUTConnector = COMMAND_LINE
SUTConnectorValue = java -jar "suts/Calculator.jar"

######################################################################
# Sequences
#
# Number of sequences and the length of these sequences
######################################################################

Sequences = 5
SequenceLength = 20
```

Figure 15: Part of the test.settings file

**hands-on 13**

Find out the role of the `.sse` file.

Change the name of the `.sse` file to the name of one of the directories in the bin folder (`\testar\bin\settings`). Now start up TESTAR again to see the effect. What if you delete the `.sse` file?
`NOTE:` Using the `.sse` file may not seem significant when running TESTAR manually. Still, it does have its importance when you want to invoke a TESTAR protocol in an automated continuous integration environment.

SECTION 6

# Customizing the TESTAR test sequences

TESTAR provides a couple of different ways to define system-specific instructions. We will go through the basic ones in this section.

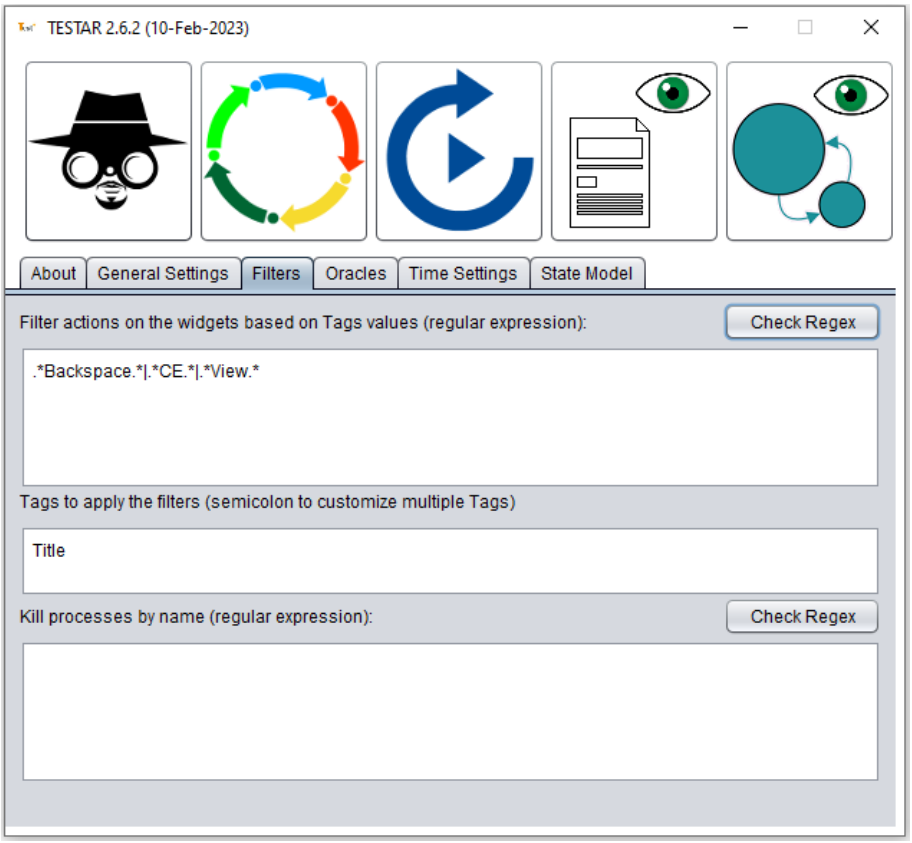
## 6.1 Adding knowledge about the SUT - Regex Action filtering

We saw before that, that from time to time, TESTAR could execute "undesirable" actions. For example, actions that minimize the SUT or even close the SUT. We can configure TESTAR to prevent the selection of these undesired actions by defining an action filter in the test settings file. The action filtering can be used to tell TESTAR *not* to take some actions during testing.

In the "Filters"-Tab you can find an input field that allows you to filter actions by defining a regular expression. TESTAR will ignore all actions that exercise control elements whose title matches the given regular expression. For example:

```
.*Backspace.*|.*CE.*|.*View.*
```

This expression will ignore clicks to all control elements whose titles contain the given strings.



**hands-on 14**

Filtering action-widgets by defining Regular Expressions (RegExs).

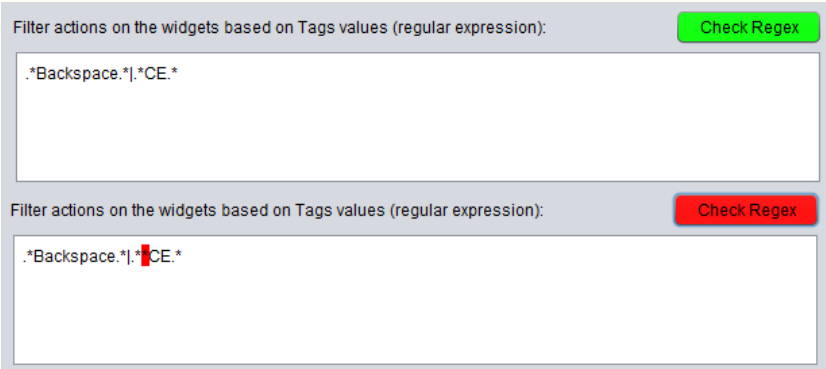
In TESTAR Settings Dialog, first, check that you still have the following protocol selected: `01_desktop_calculator`

Let's first use TESTAR SPY mode to check what actions TESTAR finds with the current settings (without any filtering). In SPY mode, all the click actions have a green dot to show where TESTAR would click to execute that action (or blinking blue text to show textual input actions).

Open the "Filters"-Tab in Settings Dialog, and write the following regular expression into the upper text area:

```
.*Backspace.*|.*CE.*
```

Click the button `Check Regex` on the right side of the filtering text area to validate that the regular expression contains a valid syntax. If it is correct, the button will become **green**, if not, the button will become **red**, and the text area will highlight the wrong regular expression character.




More instructions on using Java RegExs can be found in Appendix E.

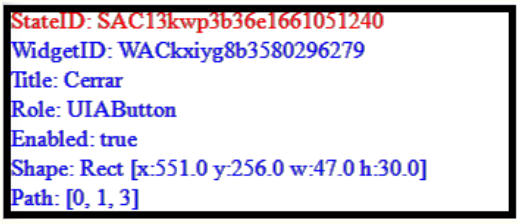
Use TESTAR Spy mode again to check that the filter works as expected - the green dots for Backspace and CE actions should have vanished:

**hands-on 15**

Filtering action-widgets by defining regular expressions.

The action-widgets we want to filter is the one that closes the SUT. Let us first use the SPY mode to find out what its title is such that we can filter that.

Using the SPY mode and hovering the mouse over the close button, you will find:




The title of the button is "Cerrar" in Spanish (or "Close" if you are using the English language).

Add the filter for this action and verify with SPY mode that the green dot has vanished. Subsequently, run TESTAR in GENERATE mode and check whether you still find failures caused by the undesirable actions that close the SUT.

## 6.2 Adding knowledge about the SUT - SPY Action filtering

You can also filter actions while in SPY mode by using the *clickfilter*-feature. CAPS_LOCK toggles the *clickfilter*-feature that enables you to filter actions by clicking on them in SPY mode. This comes in very handy when setting up your tests. Once this feature is enabled, you can just hover over the widget and press Ctrl to filter the actions on this widget from being selected during testing (you don't have to press the mouse button on the widget).

You can also undo the filtering by maintaining Shift + pressing Ctrl while hovering over a filtered widget. If the action filter you specified with a regular expression in the "Filters"-Tab was "too efficient", you can unfilter a specific action this way. Filtered actions will be stored in the file `protocol_filter.xml` that you can find in the current TESTAR's protocol folder (`\testar\bin\settings\01_desktop_calculator`).

In the next image, we can see a red square that surrounds the `Minimize` button, this is because this action-widget has been filtered by pressing Ctrl when CAPS_LOCK was toggled. Furthermore, the `Backspace` button that was previously filtered with a Regex expression now is surrounded with a green square, indicating that has been unfiltered by maintaining Shift + pressing Ctrl while CAPS_LOCK was toggled.


`IMPORTANT:` Due to problems sending the CAPS_LOCK key event to a Virtual Machine, we added an optional workaround to also be able to use the ALT key event to enable the SPY filtering mode.

**hands-on 16**

Filtering action-widgets using the clickfilter

Start the calculator in SPY mode to visualize all the green dots (those are the available, unfiltered actions). Then enable the clickfilter feature by pressing CAPS_LOCK and filter out some actions to see the effect of green dots disappearing. Subsequently, open the `protocol_filter.xml` to see the effect of what you have done.

What happens if you stop TESTAR, delete the `protocol_filter.xml` file, and start TESTAR + SPY mode again?

***TS Note:*** Due to problems sending the CAPS_LOCK event to a Virtual Machine, we added an optional workaround to also be able to use the ALT event to toggle the filtering mode. Please read the APPENDIX section D.

**hands-on 17**

Filtering action-widgets

Configure TESTAR such that it does not maximize or minimize the window anymore.

`File -> Shut Down` button is probably another action-widget you want to filter.

In addition, disallow clicks to the "Open File" menu item to prevent TESTAR from going wild on the operating system's files.

Use the SPY Mode to see whether your action-widget filters have an effect.

## 6.3 Adding knowledge about the SUT - Test Oracle

Oracles are the mechanism that tells whether a specific GUI state is correct, faulty, or suspicious. The default oracles implemented in TESTAR can find only certain types of failures. In order to detect a wider variety of failures, TESTAR allows the user to define application-specific *oracles*.

TESTAR provides different ways to define SUT-specific test oracles. Probably the easiest one, called *Suspicious Tags*, allows you to define what kind of text is considered suspicious in a Tag using regular expressions. TESTAR will use these expressions after the execution of each action in order to find potential matches with the titles of the widgets. For example:

```
.*[Ff]aultystring.*|SomeOtherFaultyString
```

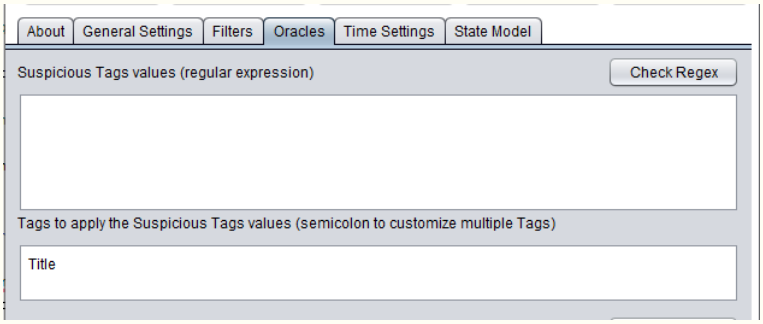
This expression will make TESTAR look for the string "Faultystring" (F letter upper- or lowercase- `[Ff]`) anywhere on the screen in any position as well as for the exact match "SomeOtherFaultyString". If TESTAR encounters such a string, it will verdict a suspicious tag output and save the corresponding sequence under:

```
\testar\bin\output\timestamp_SUTname\sequences_suspicious_tag\
```

**hands-on 18**

Adding oracles for suspicious tags using regular expressions

Select the "Oracles"-Tab in the TESTAR Settings Dialog. You will see that the Title-tag has already been configured as the default tag to check:




Now define the following regular expression into Suspicious Tags text field:

```
.*[eE]rror.*|.*[eE]xception.*
```

Click the button `Check Regex` to verify the syntax.

Run some tests again with GENERATE and check whether your test oracles detect the bugs. With correct definitions, the output folder of TESTAR should now also include **sequences_suspicious_tag** folder.

**hands-on 19**

Adding more oracles for suspicious tags using regular expressions

During handson 3, you should have found some dialogs or screens that look like a failure, in addition to crashes that TESTAR recognizes automatically. Use the suspicious tag functionality to define more test oracles that catch those.

## 6.4 Specifying oracles to detect suspicious process output

Sometimes a SUT such as the Calculator can crash and close unexpectedly or throw exception messages while running in the output and error buffers of its process. Figure 16 shows how clicking the `View -> Scientific` buttons, unexpectedly closes the Calculator as it throws an `ArithmeticException` message in the command prompt.

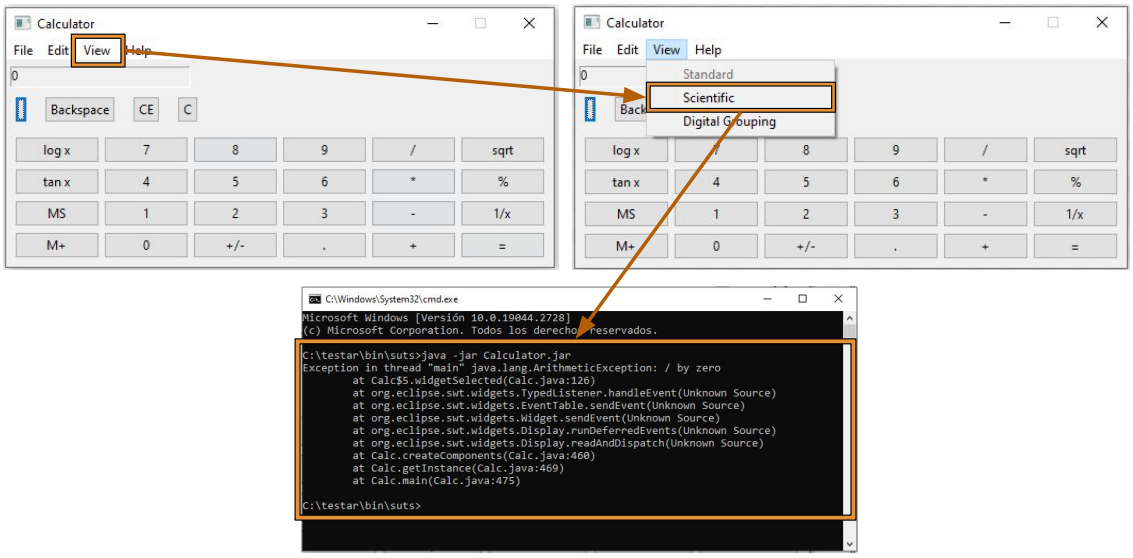
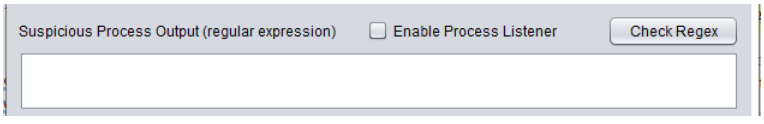



Figure 16: Calculator process error exception

TESTAR can also listen to the output and error buffers of a desktop application's process. This is useful because it allows the tester to define oracles to detect *Suspicious Process Output* by using regular expressions. For example, for catching exceptions that are printed to standard/error output but have not been otherwise properly handled.

In order to use these features, we need to enable the process listeners. This can be done in the test.settings file:
`ProcessListenerEnabled = true`

Or by activating the corresponding checkbox in the "Oracles"-tab in the TESTAR Dialogue:



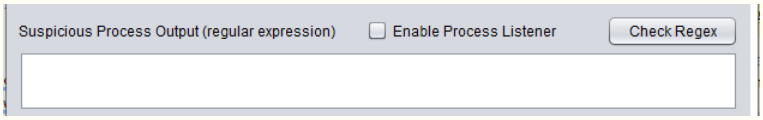

To indicate the output that should be considered suspicious, the tester has to specify the suspicious patterns with regular expressions in the same manner as the suspicious tag oracles.

Currently, these oracles based on process listeners are only enabled for desktop applications executed from the `COMMAND_LINE` option. When an oracle finds a suspicious output in the process buffer, the execution of the test is stopped.

**hands-on 20**

Adding oracles for suspicious process output using regular expressions

Use the same regular expression also for *Suspicious Process Output* in the "Oracles"-tab of the TESTAR Setting Dialog:




```
.*[eE]rror.*|.*[eE]xception.*
```

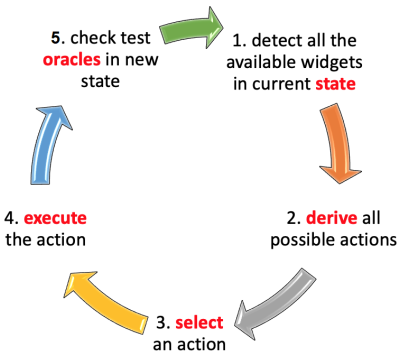
Enable the process listener by checking the box and run TESTAR in GENERATE mode again to see whether your process output test oracles detect any additional bugs.

Now the HTML report will include a message to indicate that the Verdict comes from the Process Listener, and the output process information will be stored according to its sequence in the new folder:

```
\testar\bin\output\timestamp_SUTname\logs\processListener\
```

## 6.5 Changing the way actions are derived

There are two main aspects that affect how TESTAR generates test sequences:




2. What possible actions are detected and derived from the GUI (that is defined in `derive_actions` function of TESTAR protocols), and

3. How to select from the possible actions (that is defined in `select_action` function of TESTAR protocols).

In this section, we will look at the way actions are derived. In the next section, we will look at action selection.

So far in these exercises, we have used the following method to derive actions from the widgets that are in a specific state:

`deriveClickTypeScrollActionsFromTopLevelWidgets()`

This method is called in `derive_actions` in TESTAR's protocol. It first collects the following actionable widgets, i.e., those that:

- are enabled,
- not blocked by other widgets,
- not filtered by action filters, and
- on top of the GUI, i.e., not hidden under other widgets like for example pop-ups or opened menus.

Then it derives the possible actions for these actionable widgets:

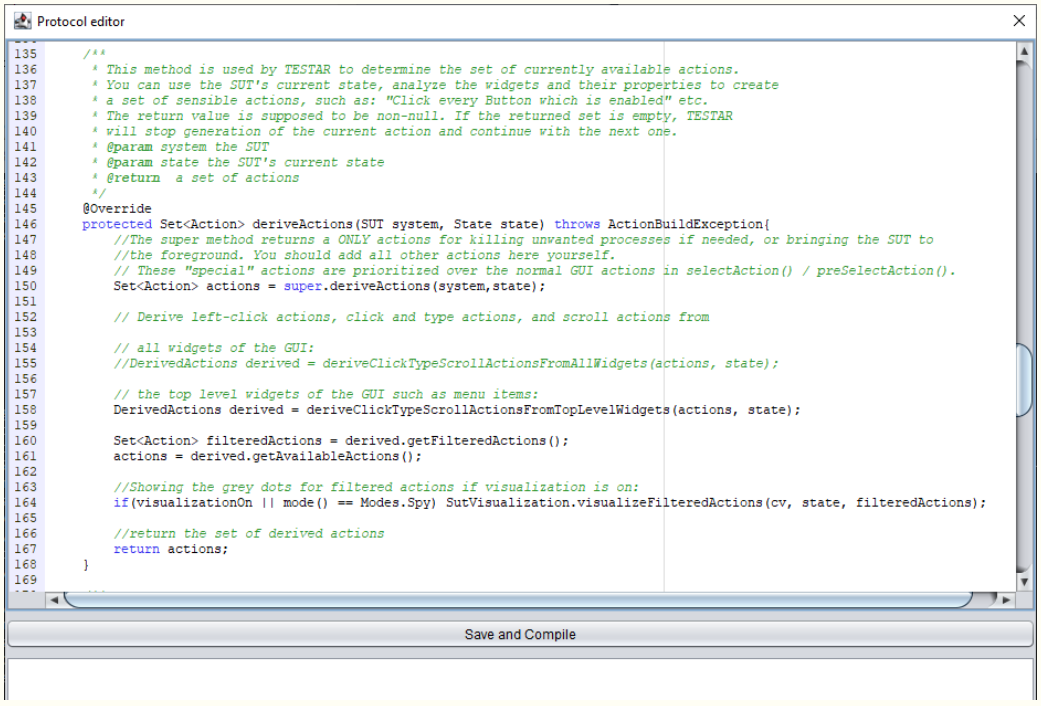
- left mouse click actions on all clickable widgets,
- pseudo-random input actions on all editable widgets, and
- random sliding actions on all widgets that have scroll bars.

**hands-on 21**

Find the code for `derive_actions` in the protocol

Remember handson 5 where we explored the TESTAR *01_desktop_calculator* protocol. Click the "Edit Protocol" button in the Dialogue. The protocol editor will open.

Browse through the code and try to find the methods that are explained above.




We can change this way of deriving actions to, for example, another function that is available in TESTAR:

`deriveClickTypeScrollActionsFromAllWidgets(actions, state)`

This function considers actionable widgets those that are:

- are enabled,
- not blocked by other widgets,
- not filtered by action filters, and

It uses these to derive the actions.

**hands-on 22**

Change the protocol to derive all actions instead of only those on top.

Click the "Edit Protocol" button in the Dialogue. The protocol editor will open. Adjust the code as needed, save, and compile the code.

Run again some tests with GENERATE and see if you can detect differences?

## 6.6 Changing the way actions are selected

TESTAR uses, by default, a random action selection algorithm (ASM) to select which action to execute from all the derived actions. This is what makes scriptless testing tools to be known as "monkey" testing tools.

Additionally to a random ASM, TESTAR implements various action selection algorithms to make the decision of *What action to select next?* smarter. Figure 17 represents how the `ActionSelectorProxy` architecture allows implementing smarter ASMs in the TESTAR to become a "smart monkey" testing tool. Let us try the `PrioritizeNewActionsSelector`, which keeps track of executed actions and compares available actions between the current and previous state, prioritizing new and least executed actions.

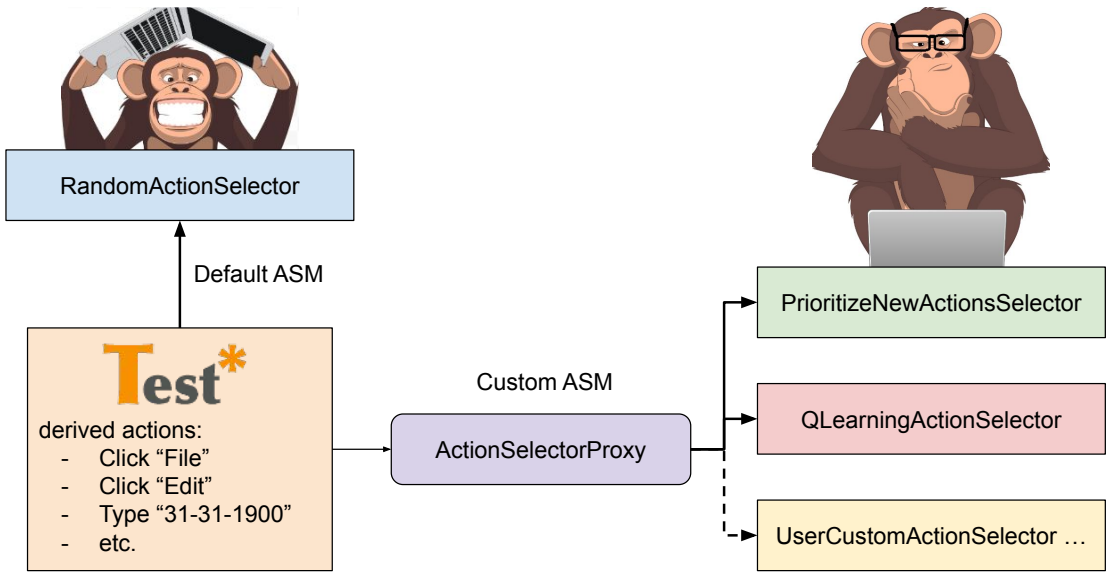



Figure 17: TESTAR action selector

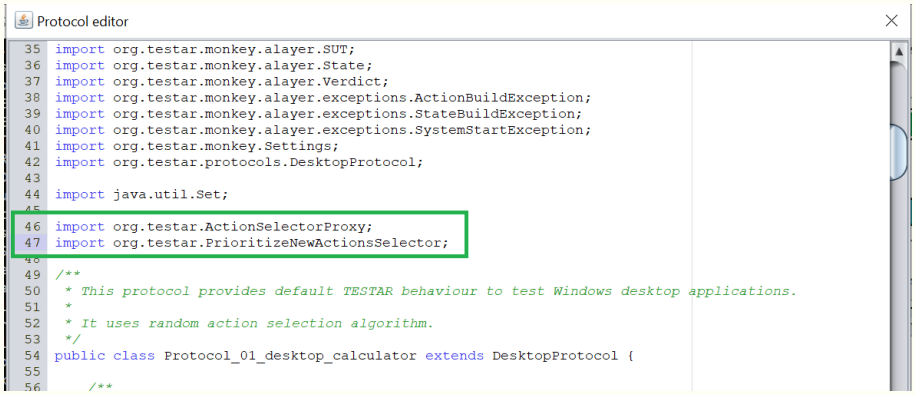
**hands-on 23**

Change the action selection algorithm

We need to initialize an action selector proxy with the smarter action selection algorithm. Then, track which actions have been derived, executed, and selected. The following changes have to implement into the TESTAR protocol:

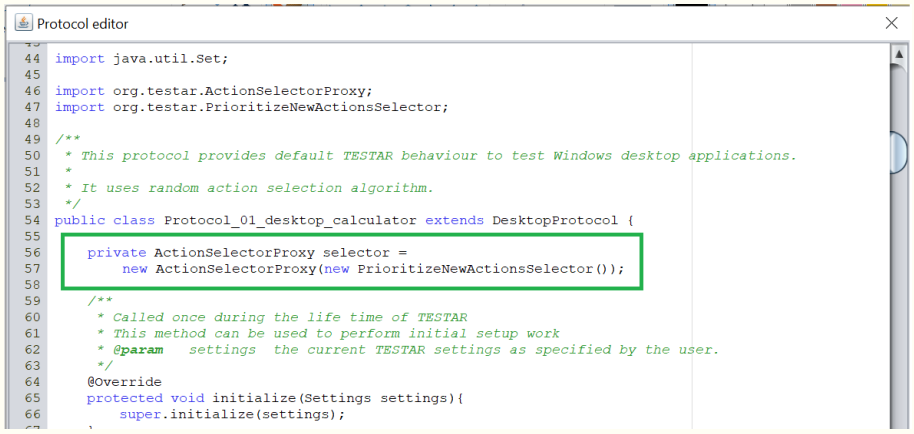
① Add the following imports:

```
import org.testar.ActionSelectorProxy;
import org.testar.PrioritizeNewActionsSelector;
```



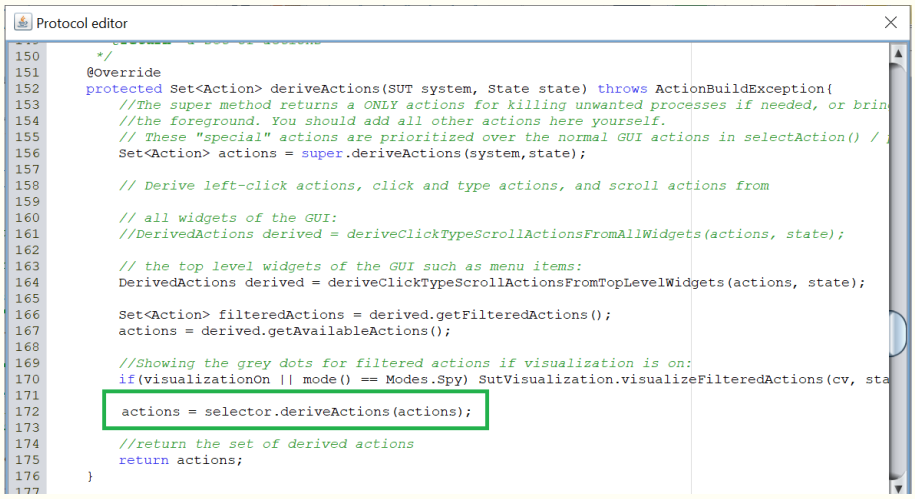

② Add the following class variable to the protocol class:

```
private ActionSelectorProxy selector =
    new ActionSelectorProxy(new PrioritizeNewActionsSelector());
```



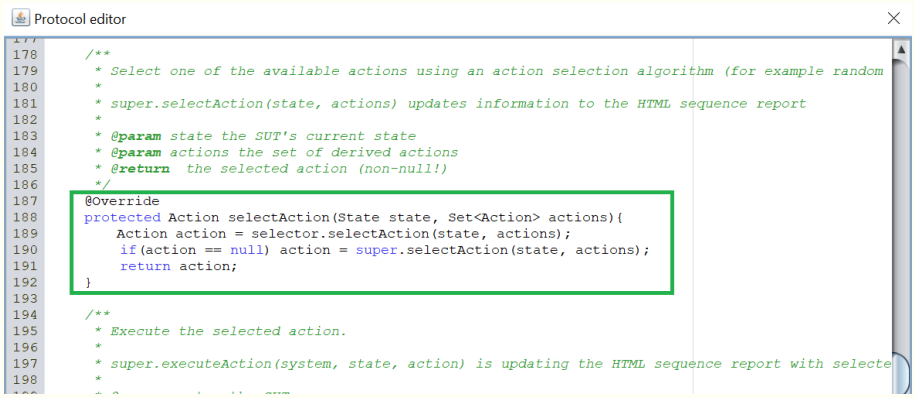

③ Add the following line to the `deriveActions()` method, just before the `return`

```
actions = selector.deriveActions(actions);
```



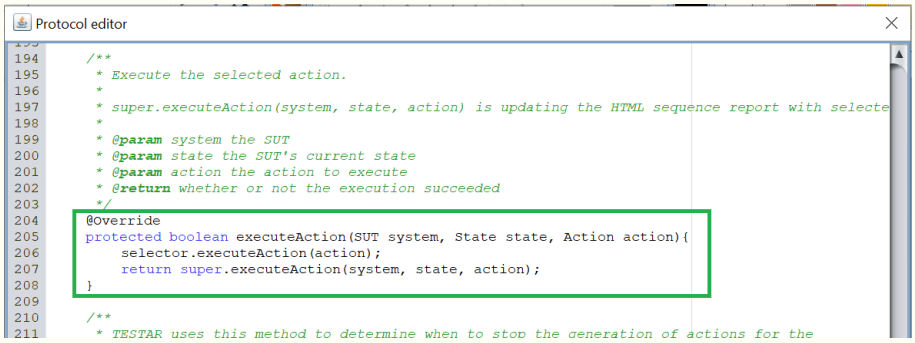

④ Replace the whole body of the `selectActions()` method with this code:

```
Action action = selector.selectAction(state, actions);
if(action == null) action = super.selectAction(state, actions);
return action;
```

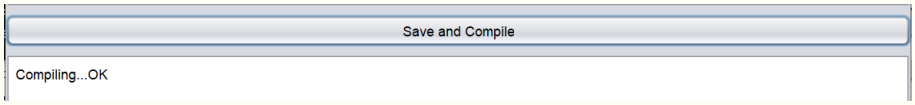



⑤ Add the following line to the `executeAction()` method, just before the `return`

```
selector.executeAction(action);
```



⑥ Click on the compile button to check the Java protocol is correct:




Run TESTAR in Generate mode with the protocol you edited (check that always compile protocol option is on). The command prompt should print some additional debugging information that helps users to track which action has been prioritized.

SECTION 7

# Connect with the System Under Test

## 7.1 Types of SUT connectors

Until now, we have connected to the SUT through the `COMMAND_LINE`. In the "General Settings"-Tab, that option was selected from the drop-down menu. In the text field we indicated that the `COMMAND_LINE`-command to connect to the SUT was:

`java -jar suts\Calculator.jar`.

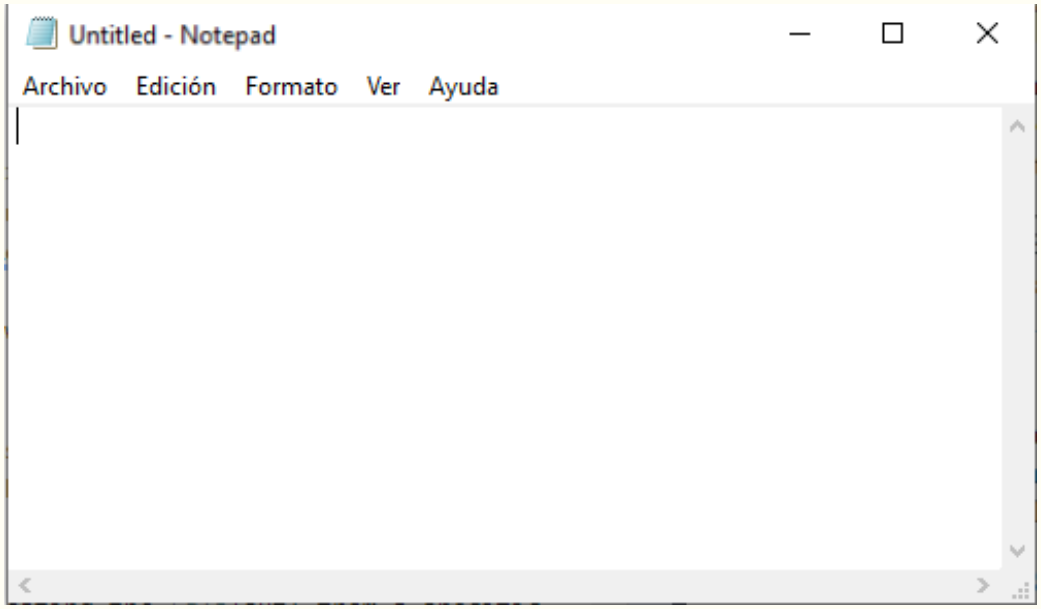
TESTAR offers a second option to connect to the SUT using the `SUT_WINDOWS_TITLE` feature. When using `SUT_WINDOWS_TITLE`, the application should already be running (started manually or started by another process) before starting TESTAR. Then TESTAR will search for an application that has a main window with the indicated title.

**hands-on 24**

Connect using the `SUT_WINDOWS_TITLE`

For this section, we are going to use Notepad as the demo desktop application.

① Launch the Windows Notepad desktop application in your system




② Launch TESTAR and change to the `desktop_generic` protocol. Change the SUTConnector dropdown to `SUT_WINDOWS_TITLE` and the value to connect to `Untitled` or another value that matches your Notepad title language.

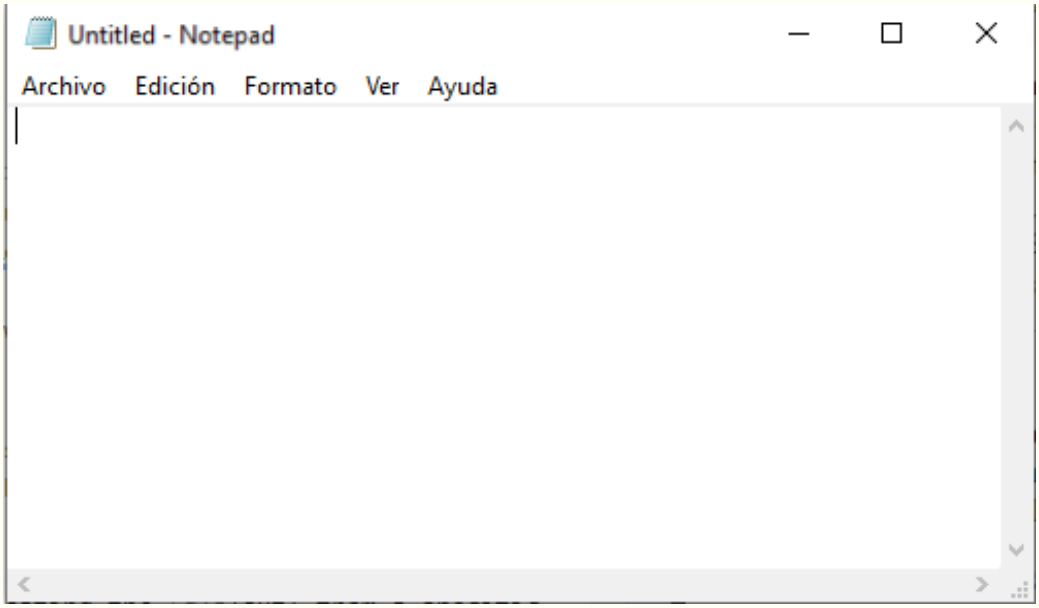

③ Run the SPY mode and verify TESTAR connects with Notepad after iterating through system desktop applications.

The third option is to connect through the `SUT_PROCESS_NAME`. When using `SUT_PROCESS_NAME`, the application should already be running (started manually or started by another process) before starting TESTAR. When using `SUT_PROCESS_NAME` as the `"SUT Connector"`, TESTAR will search for a process with a process name matching the executable of the SUT.

**hands-on 25**

Connect using the `SUT_PROCESS_NAME`

① Launch the Windows Notepad desktop application in your system




② Launch TESTAR and change to the `desktop_generic` protocol. Change the SUTConnector dropdown to `SUT_PROCESS_NAME` and the value to connect to `notepad.exe` process.


③ Run the SPY mode and verify TESTAR connects with Notepad. This connector mode `SUT_PROCESS_NAME` is faster than the previous `SUT_WINDOWS_TITLE`.

To test Java applications, we do not recommend using the `SUT_PROCESS_NAME` connector. This is because the process name of a Java application is `java.exe` or `javaw.exe` and hence cannot be easily matched to find the correct SUT executable.

## 7.2 Execute TESTAR from the command line

TESTAR allows its execution and settings configuration from the command line. This is needed when we want to add TESTAR to a Continuous Integration (CI) pipeline. By default is executed with the selected protocol (.sse file) and the test.settings values of that protocol.

From the command line, it is also possible to select the desired protocol to execute TESTAR and change the values of the test.settings. The protocol to be executed can be selected using the "sse" parameter next to the name of the desired protocol. For example:

```
testar sse=desktop_generic
```

Other settings are input using the pairs "parameterX=valueX" separated by space. For example:

```
testar ShowVisualSettingsDialogOnStartup=false Mode=Generate
```

Some of the most interesting parameters that can help to integrate TESTAR as a Continuous Integration tool are:

`sse`: to select the desired protocol

`ShowVisualSettingsDialogOnStartup`: To run TESTAR without the dialog (this can be considered mandatory)

`Mode`: TESTAR execution Mode (mostly used for CI is GENERATE)

`SUTConnector` and `SUTConnectorValue`: The way to link with the desired application to be tested

`Sequences` and `SequenceLength`: The number of iterations and actions that TESTAR

will execute

NOTE: Certain characters, such as slashes or quotation marks, must be entered in a double way to respect the treaty of special characters. For example:

```
SUTConnectorValue=" ""C:\\Windows\\notepad.exe"" "
```

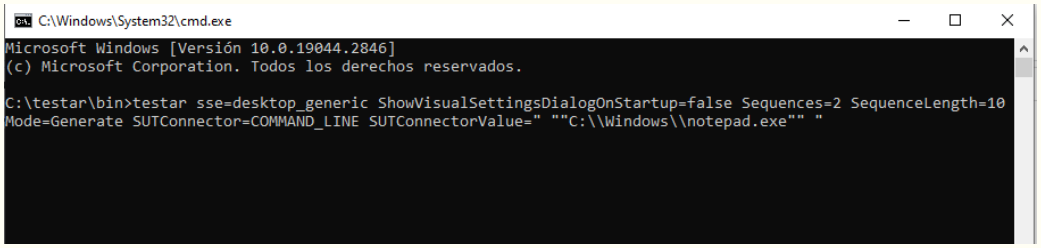
**hands-on 26**

Simulate a Continuous Integration execution to test Notepad with TESTAR

① Close TESTAR and open a command prompt in the `\testar\bin\` directory

② Type or paste the following settings in the command prompt. You need to paste everything into one command line without line breaks.

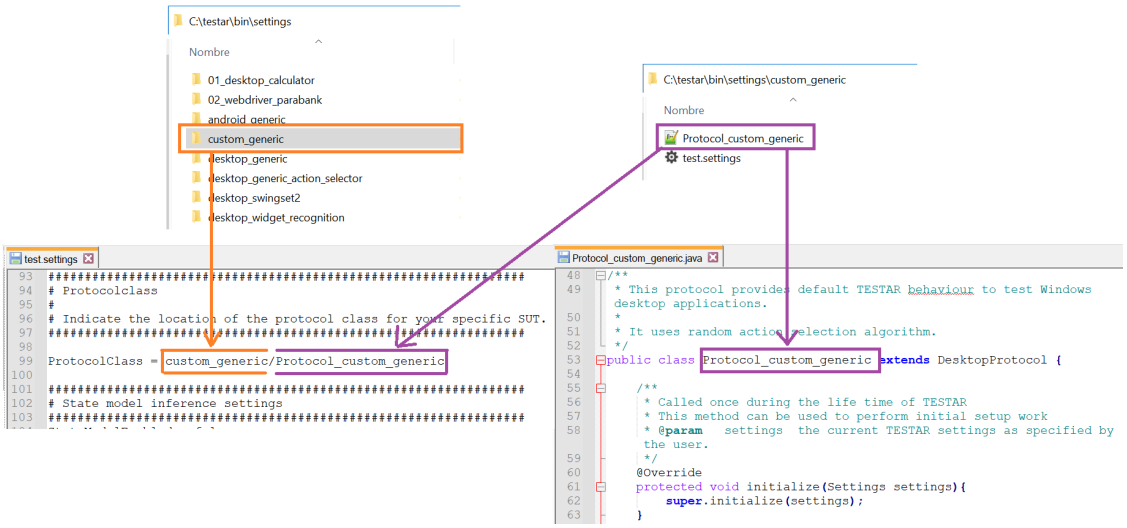
```
testar sse=desktop_generic ShowVisualSettingsDialogOnStartup=false
    Sequences=2 SequenceLength=5 Mode=Generate SUTConnector=COMMAND_LINE
    SUTConnectorValue=" ""C:\\Windows\\notepad.exe"" "
```




Observe how TESTAR automatically runs without opening the dialog.

## 7.3 Create a custom protocol

Sometimes it is necessary to create new TESTAR protocols to configure the settings and Java class in order to test different SUTs or to prepare multiple protocols to test different parts of the same SUT. The only requirement to custom a new protocol is to create a new folder with a Java and `test.settings` files inside the `testar\bin\settings` directory. Then, update the name of the protocol in the Java class name + `ProtocolClass` setting variable.

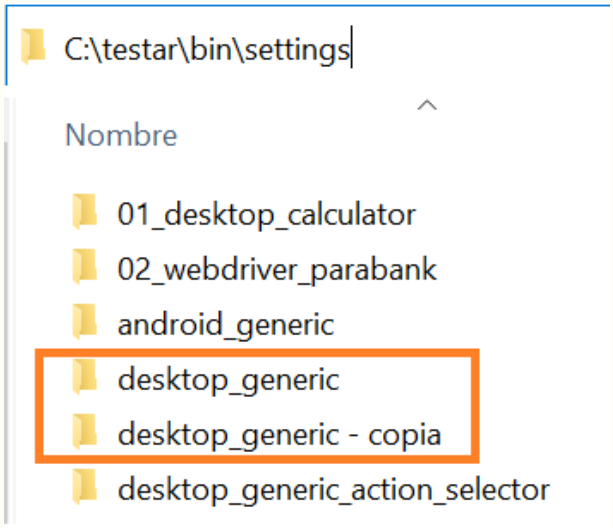



**hands-on 27**

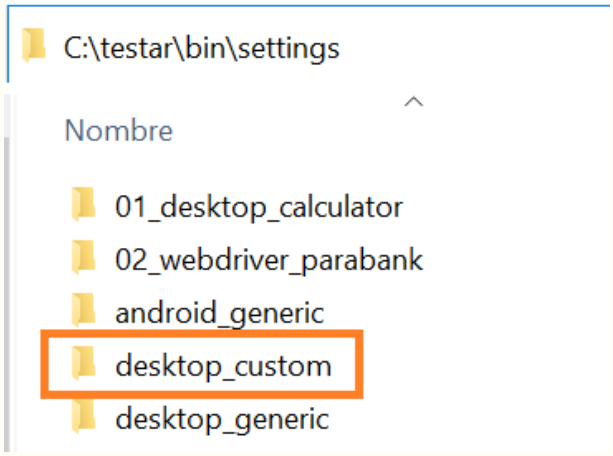
Create a new TESTAR protocol

① Close TESTAR if running.

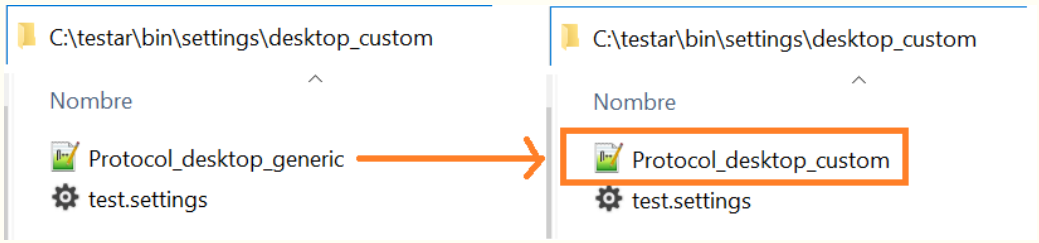
② Open the `testar\bin\settings` directory and copy paste the `desktop_generic` folder.



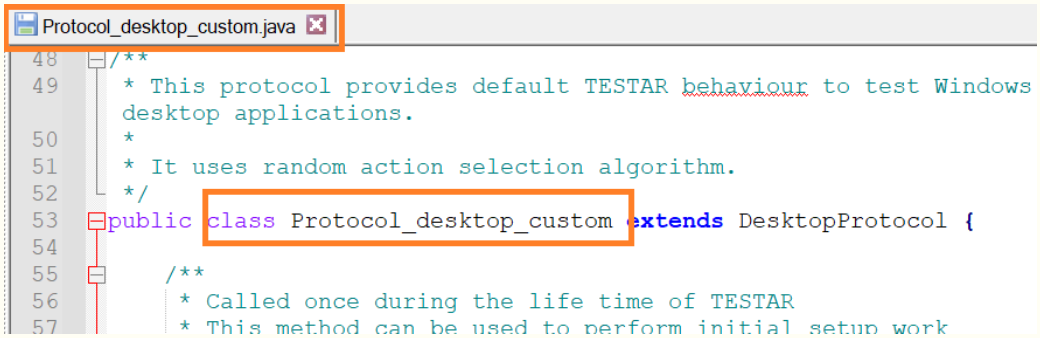

③ Rename the new folder with `desktop_custom` name.



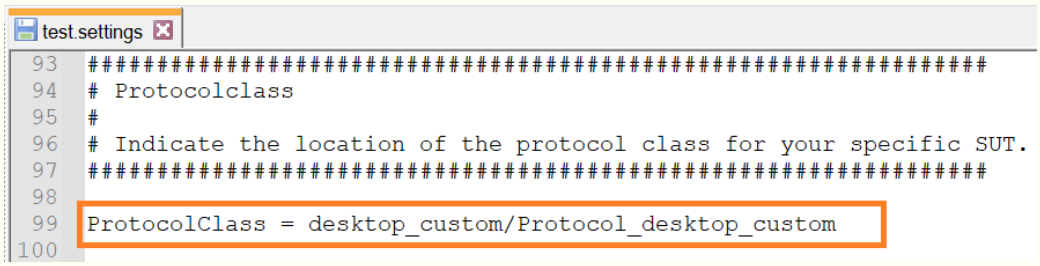

④ Open the folder `desktop_custom` and rename the `Protocol_desktop_generic` protocol name with `Protocol_desktop_custom` name.



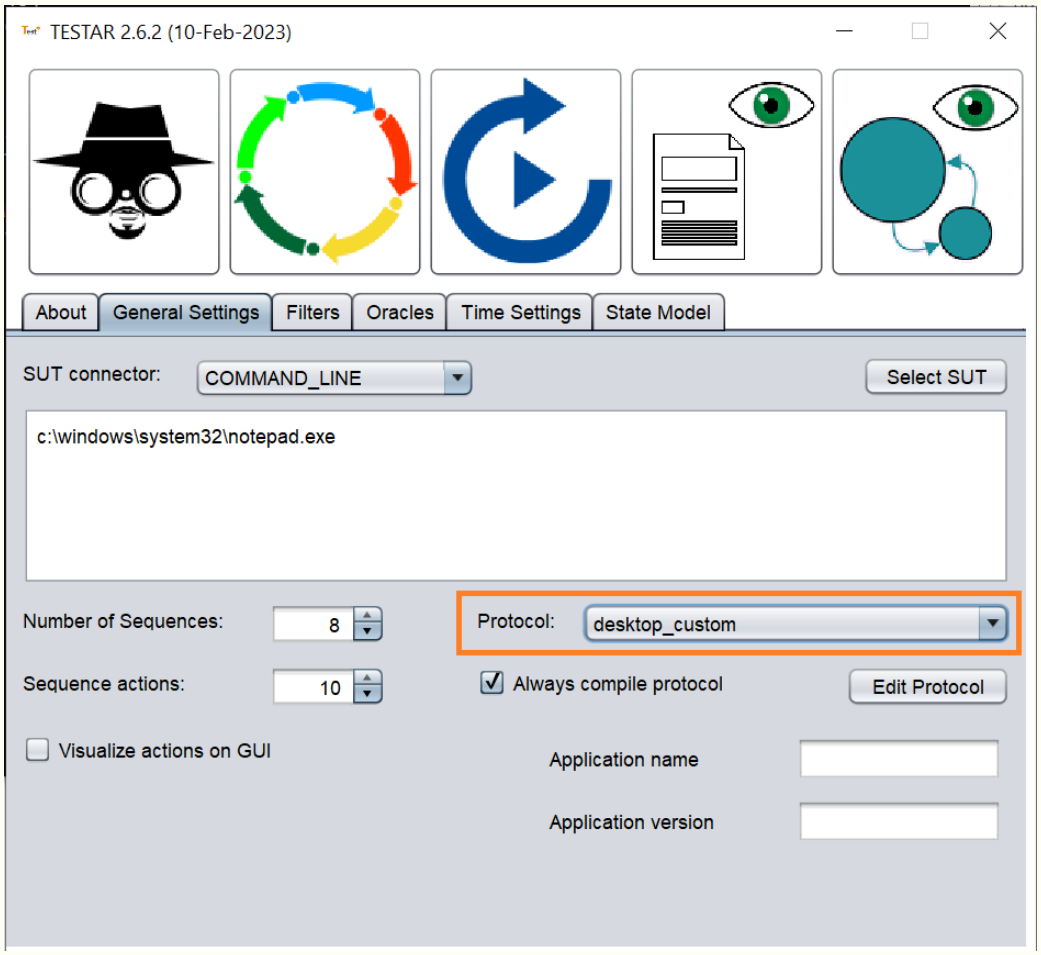

⑤ Open the Java class `Protocol_desktop_custom` and edit the class name to match the filename.




⑥ Open the `desktop_custom\test.settings` file and edit the `ProtocolClass` setting variable to match the custom folder and Java class.



⑦ Run TESTAR and verify that the new `desktop_custom` protocol can be selected.



SECTION 8

# Testing web applications with TESTAR

In addition to Windows accessibility API, TESTAR supports using Selenium WebDriver[4] for testing web applications.

The TESTAR webdriver inspects the DOM and extracts the information via a (JS) web extension, added to the browser. The Selenium webdriver acts as a bridge between the browser and the webdriver module of TESTAR.

Apart from the improved detection of widgets on web applications, using the wedriver module allows the user to use TESTAR 'natively'. As all interactions are on the browsers viewport, it is not necessary to use a VM. The webdriver version of TESTAR is tested on 3 mayor platforms: Windows 10, OS X, and Linux. It has been tested with Chrome/Chromium, Firefox, and Edge (Windows). That said, using Chrome seems to be the best option.

We will use an open-source web application Parabank (`https://parabank.parasoft.com/`) as a SUT. It provides a new challenge for a test monkey - a login page that requires a valid username and password.

We have prepared another TESTAR protocol for this exercise:

`02_webdriver_parabank`

## 8.1 Installing the Selenium Webdriver

For TESTAR to be able to connect to a browser, it needs the Selenium Webdriver. It acts as a bridge between TESTAR and the browser of your system. It is a W3C standard, and is supported by the 3 biggest browsers : Chrome/Chromium, Edge and Firefox. Unless there is a specific requirement for another browser, the driver for Chrome is recommended as it has the best performance.

The drivers can be found here:

- Chrome/Chromium
  `https://chromedriver.chromium.org/downloads`
- Firefox (Geckodriver)
  `https://github.com/mozilla/geckodriver/releases`
- Edge (Microsoft Webdriver)
  `https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/#downloads`

In the indicated URLs, the different versions of the webdrivers are updated. When we are going to select a version to download, it is important to consider the current version of our browser (`chrome://settings/help`).

`NOTE:` If the versions are incompatible and you need to download a new chromedriver, TESTAR will display a warning message.

[4] `https://www.selenium.dev/`

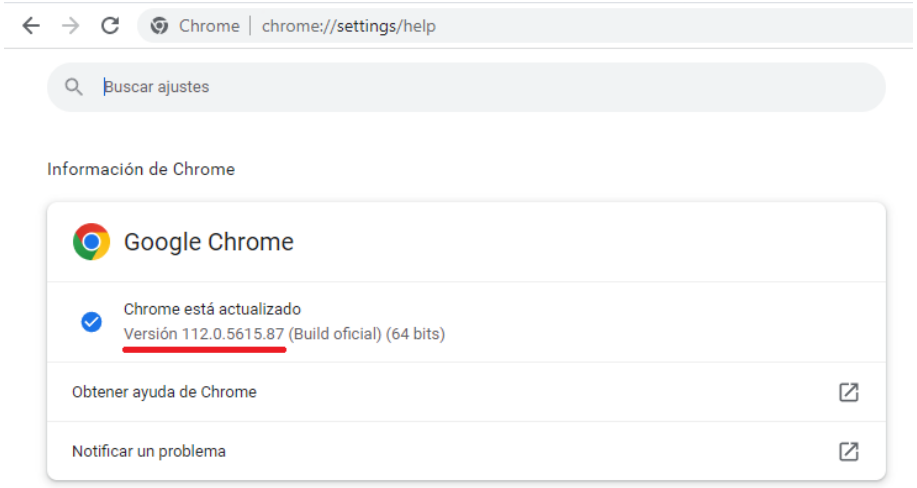



Figure 18: Chrome Browser version

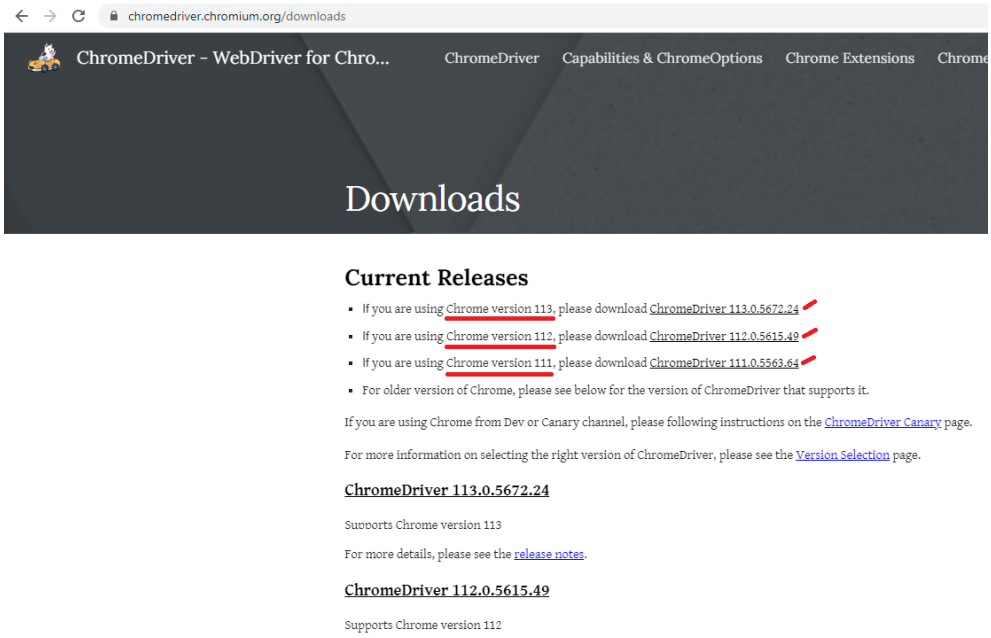



Figure 19: Download Selenium ChromeDriver

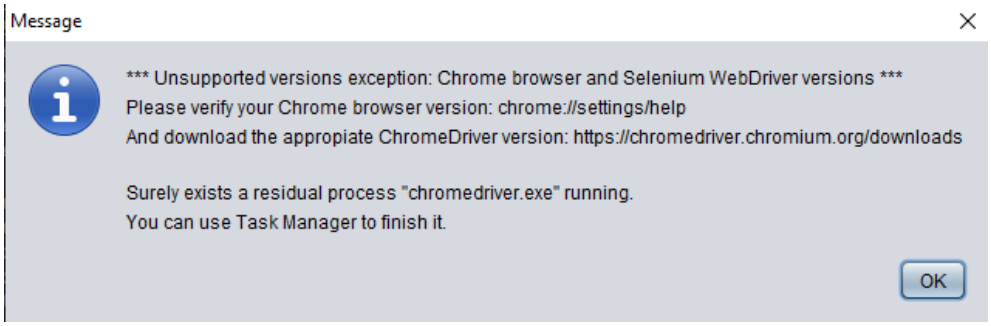



Figure 20: Warning message: ChromeDriver version error

For all variants, it is important to download the version that belongs to the browser installed on your system. The location where you install the driver is important, as it needed in the TESTAR configuration. Sensible locations would be :

- Windows : `C:\Windows\chromedriver.exe`
- OS X : `/usr/local/bin/chromedriver`
- Linux : `/usr/lib/chromium-browser/chromedriver`

**hands-on 28**

Install the Selenium Webdriver for Chrome on your system

Get the Chrome/Chromium from here `https://chromedriver.chromium.org/downloads` (see Figure 19). Remember to check the current version of your browser (`chrome://settings/help`).

## 8.2 Settings for TESTAR to test web applications

In TESTAR Settings Dialog, when the selected protocol is:

`02_webdriver_parabank`

The SUT connector is:

`WEB_DRIVER`

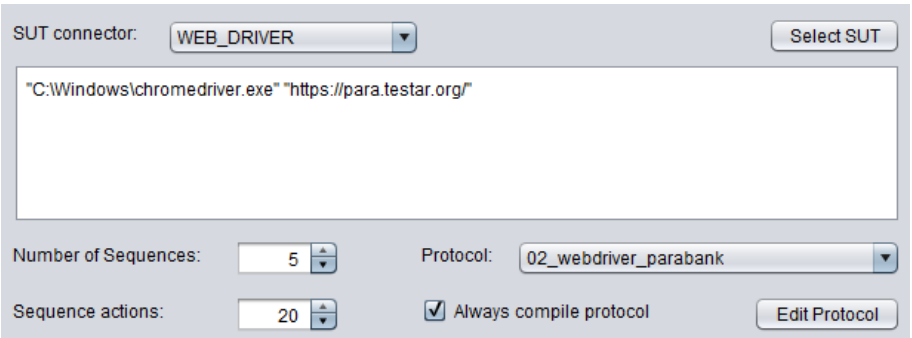
And in the SUT connector value, you can see the start-up path is written as:

```
"C:\Windows\chromedriver.exe" "https://para.testar.org/"
```

This means:

- Selenium webdriver path: indicates the PATH towards the location of the Selenium Webdriver. If this differs from the path you used in handson 28, change this here.
- URL of the web application you want to test.

The ChromeDriver is started up with the URL of the website as one of its arguments.



**hands-on 29**

Start up the web application in SPY mode

Make sure the path to ChromeDriver is correct, and its version has to match the version of the Chrome web browser. Use TESTAR SPY mode to check if TESTAR starts WebDriver correctly and finds actions on the web application. Check also whether the action filtering is sufficient.


**hands-on 30**

Run some tests for the web application.

Run TESTAR GENERATE-mode to see how it works. You will see that without providing the username and password, you will not get far with the login process. You might find a failure with *Suspicious Tags* oracle if TESTAR tries to login with invalid input, but if you look into it, it is probably a false positive (test fails but SUT does not have a bug). In this case, you might want to remove "Error" from the suspicious tags oracle.

## 8.3 Adding knowledge about the SUT - specific input actions

TESTAR protocols allow triggering pre-specified actions in a specific state of the SUT. This is useful for inputting login credentials or navigating to a specific part of the application before starting the testing.

To make this implementation easier for the users, there are two pre-defined methods for executing actions, which execute actions such as click or type on a widget with a specific Tag value. These methods can be found in `GenericUtilsProtocol.java`[5]. We will explain them below.

```
// Click pre-defined action
waitAndLeftClickWidgetWithMatchingTag(Tags.Title, // Tag Name
                                      "Next", // Tag Value to find
                                      state, // State to search
                                      system, // System to interact
                                      5, // maxNumberOfRetries
                                      1.0); // waitBetweenRetries
```

The `waitAndLeftClickWidgetWithMatchingTag` method will try to find a widget in state `state` that has a tag `Tags.Title` with the exact value `Next` and then click on that. The variable `system` is passed to the method to be able to execute the click action.

When interacting with an application, sometimes TESTAR executes actions so quickly that the GUI doesn't have time to load all widgets completely. For this reason, this `waitAndLeftClickWidgetWithMatchingTag` method internally tries to match the Tag and Value to perform the action a `maxNumberOfRetries` and `waitBetween` these tries some seconds. In this case, we indicate we want to try this click action `5` times and wait `1` second between each try. This helps to create a more robust GUI-triggered action.

```
// Click and type pre-defined action
waitLeftClickAndTypeIntoWidgetWithMatchingTag(Tags.Title, // Tag Name
                                              "username", // Tag Value to find
                                              "testar", // Text to type
                                              state, // State to search
                                              system, // System to interact
                                              5, // maxNumberOfRetries
                                              1.0); // waitBetweenRetries
```

The `waitLeftClickAndTypeIntoWidgetWithMatchingTag` method will try to find a widget in state `state` that has a tag `Tags.Title` with the exact value `username` and then Type the text `testar`.

Typing in text fields requires TESTAR to interpret each char of a string as a Keyboard Key (KBKey) and translate these KBKeys chars into code events. However, special characters are sometimes challenging to type due to the Keyboard language differences. For this reason, we have implemented a Paste in text fields action that copies the desired string into the system clipboard and then pastes it into the text field widget.

Using the `waitLeftClickAndPasteIntoWidgetWithMatchingTag` method, TESTAR will try to find a widget in state `state` that has a tag `Tags.Title` with the exact value `username` and then Paste the text `testar`.

[5] https://github.com/TESTARtool/TESTAR_dev/blob/master/testar/src/org/testar/protocols/GenericUtilsProtocol.java

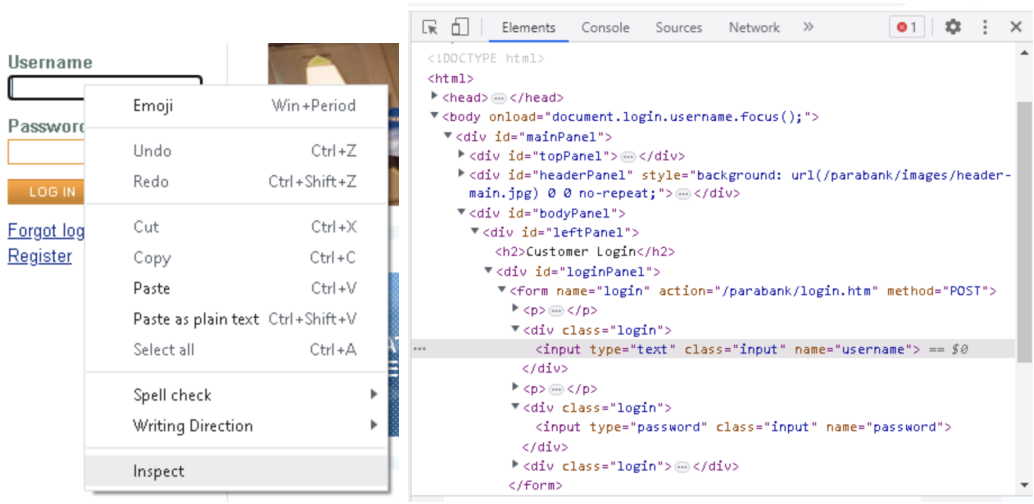
## 8.4 Adding knowledge about the SUT - login to parabank

In the case of Parabank, the login fields are visible directly when the Parabank application is started, i.e., the initial state. In this case, we can use the TESTAR's protocol method `beginSequence()` that is executed before TESTAR starts generating the test sequence. Note that `beginSequence()` is not executed when using SPY mode.

Let us use the methods explained in the previous section to define login actions for Parabank. A valid login is:

username: *john*
password: *demo*

Since Parabank is a web application, you can spy it with Chrome web browser's Inspect facility. Right-click on the Username text field, and select *Inspect* to see the widget parameters and find the information about the tags you need.




We use these parameters to define TESTAR configuration to do the login actions. Remember that for desktop applications, we have used TESTAR SPY-mode to check these tags and parameters.

The web attribute `name` is mapped as `WdTags.WebName`. You can find more information about these TESTAR WebDriver Tags in the class `WdTags.java`[6].

**hands-on 31**

Edit the protocol to enable the login

Check that you have `02_webdriver_parabank` protocol selected and press *Edit Protocol* button in TESTAR Settings Dialog. That should open TESTAR protocol editor and you should see Java code in it.

(You can also edit the protocol by using a Java IDE or text editor, the file can be found in the folder:

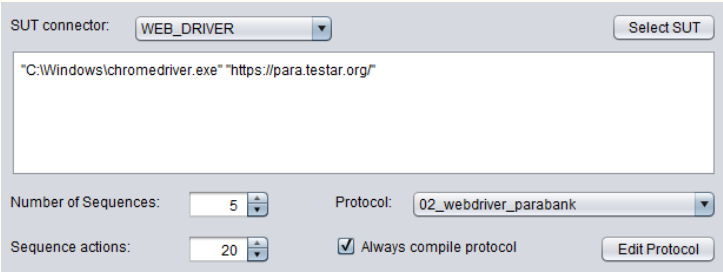
[6] `https://github.com/TESTARtool/TESTAR_dev/blob/master/webdriver/src/org/testar/monkey/alayer/webdriver/enums/WdTags.java`

`testar/bin/settings/02_webdriver_parabank`

Find the `beginSequence` method from the code. Copy and paste the following snippet of code into the body of the method (un-comment the code if it is already there by removing `/*` and `*/`):

```
// write "john" to username text field:
waitLeftClickAndTypeIntoWidgetWithMatchingTag(WdTags.WebName,
                                              "username",
                                              "john",
                                              state,
                                              system,
                                              5,
                                              1.0);

// write "demo" to password text field:
waitLeftClickAndTypeIntoWidgetWithMatchingTag(WdTags.WebName,
                                              "password",
                                              "demo",
                                              state,
                                              system,
                                              5,
                                              1.0);

// click login-button:
waitAndLeftClickWidgetWithMatchingTag(WdTags.WebValue,
                                      "Log␣In",
                                      state,
                                      system,
                                      5,
                                      1.0);
```

Click `Save and Compile` in the Protocol Editor, or check that *Always compile protocol* checkbox is enabled in TESTAR "General Settings"-tab.

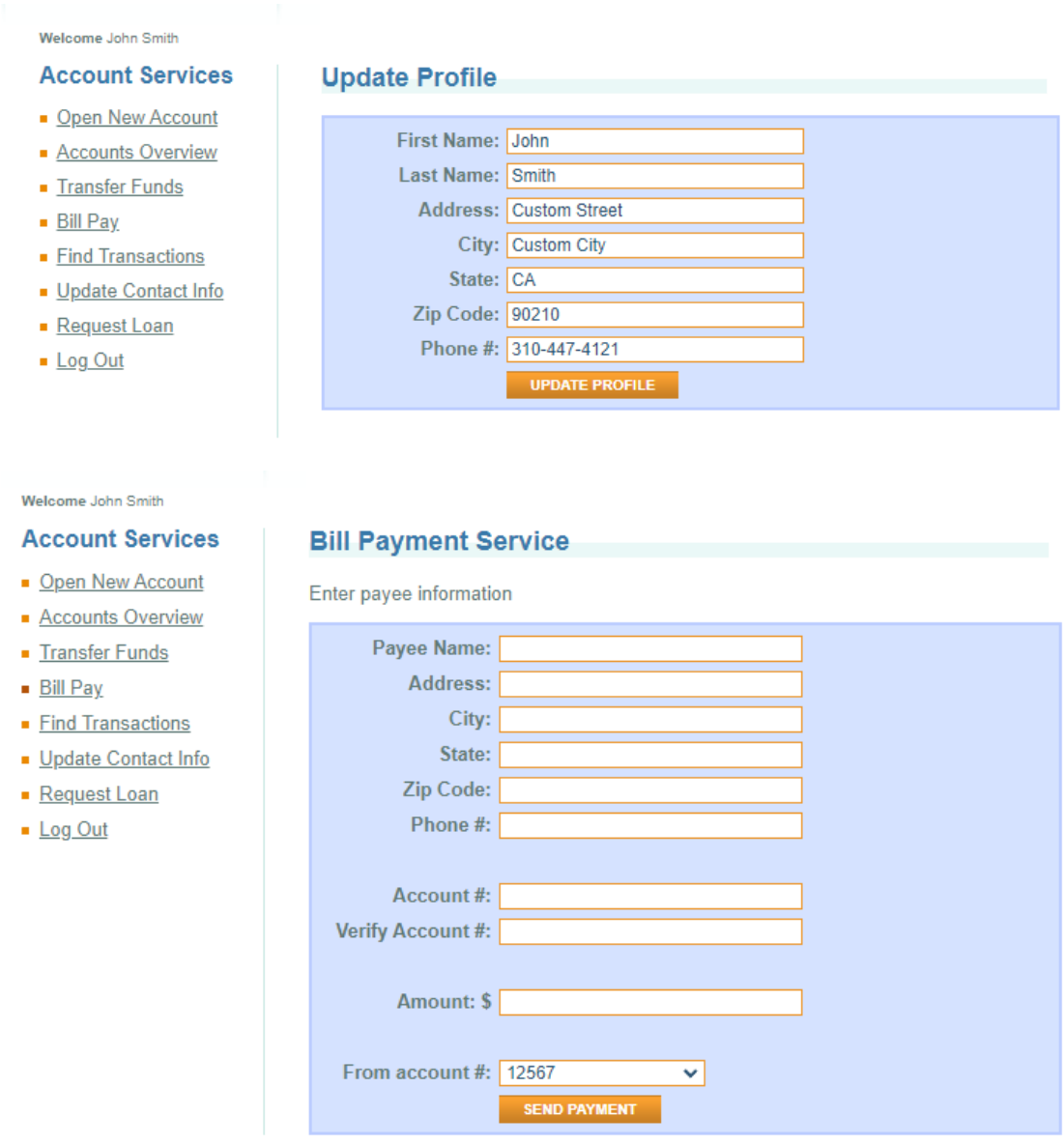



Run TESTAR in Generate mode and check whether the added code manages to login into Parabank application.

## 8.5 System specific input actions - deriveActions()

Sometimes, the state that requires a specific input is not visible right away when the system starts. Then we need to define a unique way of detecting the state of SUT that needs the input and then trigger pre-specified actions that fill in the data. After that, TESTAR continues the automated testing.

For the previous Parabank login example, we always executed a trigger action login when the web started. However, these `Update Profile` or `Bill Payment Service` are states that may occasionally appear in the testing process.




In `deriveActions()`, the idea is to check if the form itself or the first element of the form exists, then create a compound action that triggers the execution of multiple click and type actions until all form widgets are interacted.

We can implement this compound action in two ways:

1. Create a compound action in `deriveActions()` and add it to the possible actions to execute. Then TESTAR will have the possibility to execute it, but it won't always happen.
2. Only return the compound action in `deriveActions()`. Then TESTAR will always execute it because it will be the only action available.

**hands-on 32**

Use ParaBank web application as an example to create a trigger for form filling.

① Let's first use the inspect facility of the web browser on the web site we want to add the trigger.

- Open a web browser and go to `https://para.testar.org/`
- Login with username=john and password=demo
- Click on Update Contact Info
- Right-click on First Name text field and select Inspect
- You should get the following info:

```
<input id="customer.firstName"
name="customer.firstName"
class="input␣ng-pristine␣ng-valid␣ng-not-empty␣ng-touched"
type="text"
ng-model="customer.firstName">
```

- Let's try using `id="customer.firstName"` as the trigger to recognize the state.

② Start TESTAR and check that you have the `02_webdriver_parabank` protocol selected and press "Edit Protocol"-button in TESTAR Settings Dialog. That should open the TESTAR protocol editor, and you should see Java code in it. You can also edit the protocol by using a Java IDE or text editor. The file can be found in this folder:

`testar/bin/settings/02_webdriver_parabank`

③ Find the `deriveActions` method from the code. Copy and paste the following snippet of code into the body of the method (un-comment the code if it is already there by removing `/*` and `*/`):

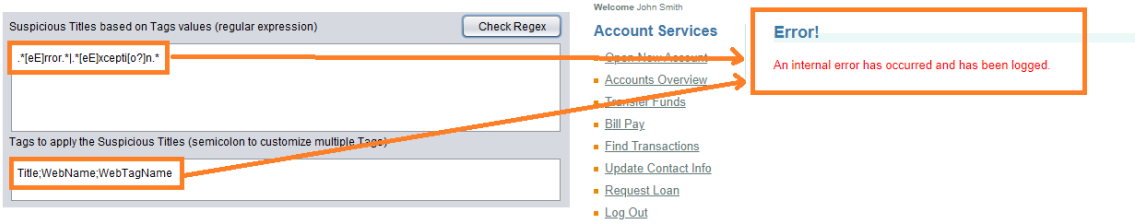
```
// Check if the update profile element is found:
Widget nameWidget = getWidgetWithMatchingTag(
    "id", "customer.firstName", state);

if(nameWidget != null){
  // Update profile found, create and return the triggered action:
  // Create a compound action to include multiple actions as one:
  CompoundAction.Builder multiAction = new CompoundAction.Builder();

  // Action to type text into the Name field:
  multiAction.add(ac.clickTypeInto(
    nameWidget, "Triggered␣Name", true), 1.0);

  // Action to type text into id="customer.lastName":
```

```
  Widget lastNameWidget = getWidgetWithMatchingTag(
    "id", "customer.lastName", state);
  multiAction.add(ac.clickTypeInto(
    lastNameWidget, "Triggered␣Last␣Name", true), 1.0);

  // Action to type text into id="customer.address.street":
  Widget streetWidget = getWidgetWithMatchingTag(
    "id", "customer.address.street", state);
  multiAction.add(ac.clickTypeInto(
    streetWidget, "Triggered␣Street", true), 1.0);

  // Action to type text into id="customer.address.city":
  Widget cityWidget = getWidgetWithMatchingTag(
    "id", "customer.address.city", state);
  multiAction.add(ac.clickTypeInto(
    cityWidget, "Triggered␣City", true), 1.0);

  // You can add here more form widgets

  // Action on Update Profile button, value="Update Profile"
  Widget submitWidget = getWidgetWithMatchingTag(
    "value", "Update␣Profile", state);
  multiAction.add(ac.leftClickAt(submitWidget), 1.0);

  // Build the update profile compound action
  Action updateProfileAction = multiAction.build();

  // Returning a list of actions having only the updateProfileAction
  return new HashSet<>(Collections.singletonList(updateProfileAction));
}
```

④ Run TESTAR in Generate-mode until you see that TESTAR clicks on `Update Profile` and your triggered action is executed.

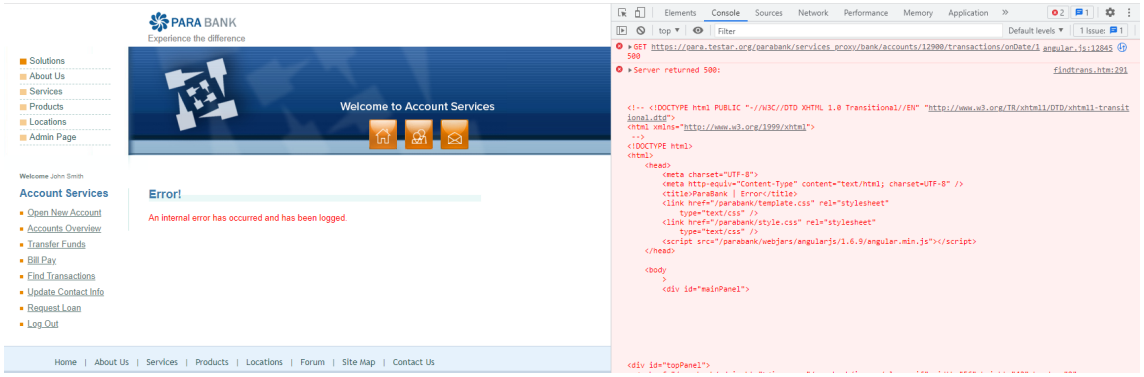
## 8.6 Webdriver oracles to detect suspicious browser console messages

As we have learned with Desktop applications, TESTAR is able to detect *Suspicious Tags* by matching regular expressions in the GUI widgets properties. This can allow users to find some Parabank states with `Error` messages when running TESTAR.

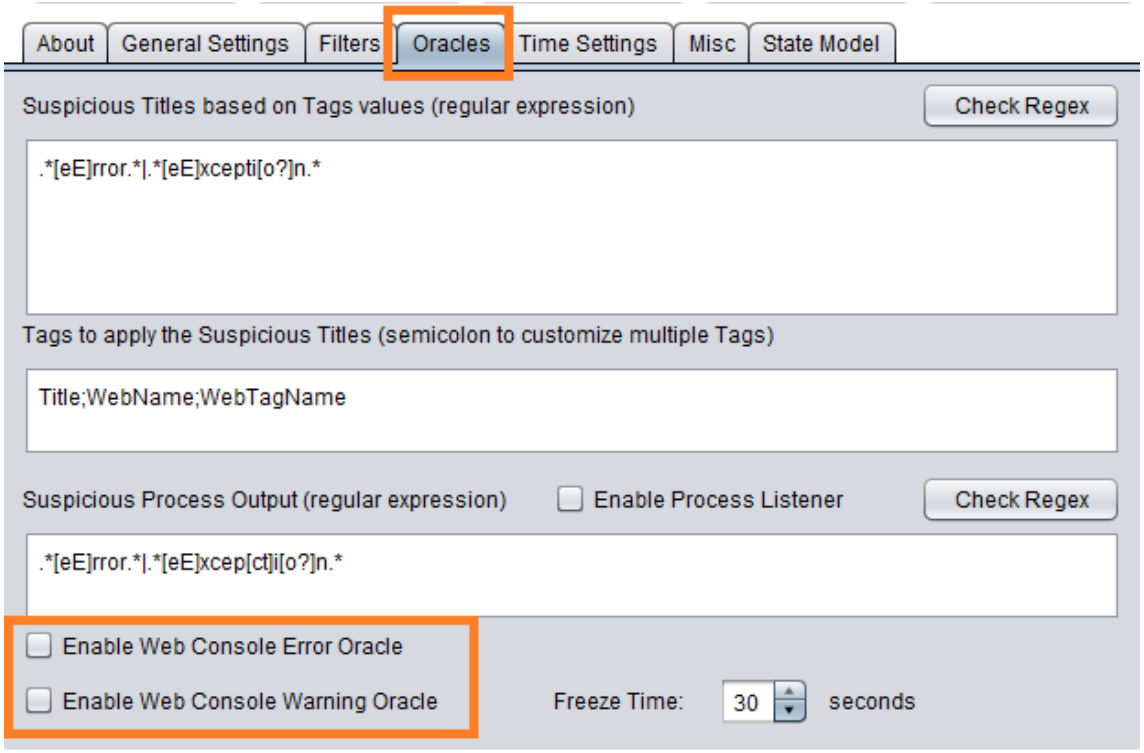



Similar to Desktop applications, on which TESTAR offers the possibility to read the process output buffers to match regular expressions, for Web applications, TESTAR contains

a feature to read the console of the browser to detect erroneous and warning messages. Obtaining this information can help detect issues on the web page that are not easy to detect from the GUI or enrich the information of the GUI errors with additional console error information.




From the Oracle-Tab in the TESTAR dialog, users can enable or disable this feature that allows detecting ALL error or warning web console messages by default. In case users desire to apply some regular expressions to detect only specific messages instead of all of them, they can edit the test.settings file to add matching patterns to the error and warning consoles individually.



```
#################################################################
# WebDriver Browser Console Oracles
#
# WebConsoleErrorOracle: Enable or Disable applying ORACLES to the browser error console
# WebConsoleErrorPattern: Regular expressions ORACLE to find suspicious messages in the browser error console
# WebConsoleWarningOracle: Enable or Disable applying ORACLES to the browser warning console
# WebConsoleWarningPattern: Regular expressions ORACLE to find suspicious messages in the browser warning console
#################################################################

WebConsoleErrorOracle = false
WebConsoleErrorPattern = .*.*
WebConsoleWarningOracle = false
WebConsoleWarningPattern = .*.*
```

**hands-on 33**

Run some Parabank tests to detect all error console messages

Run TESTAR and select the protocol `02_webdriver_parabank`. Change to the Oracle tab in the dialog and enable the `Web Console Error Oracle` checkbox.

Start testing sequences with the Generate mode of TESTAR and open the HTML reports checking if some console error messages have been detected.

**hands-on 34**

Run some Parabank tests to detect specific warning console messages

Open the test.settings file that exists in the parabank protocol folder:

`testar/bin/settings/02_webdriver_parabank/test.settings`

First, find the `WebConsoleErrorOracle` setting and disable it. Then, find the `WebConsoleWarningOracle` setting and enable the feature with the `true` option. Finally, edit the next `WebConsoleWarningPattern` setting to add a regular expression that matches the `Wrong` values.

```
WebConsoleErrorOracle = false
WebConsoleErrorOracle = .*.*
WebConsoleWarningOracle = true
WebConsoleWarningOracle = .*[wW]rong.*
```

Start testing sequences with the Generate mode of TESTAR and open the HTML reports checking if some console warning messages with the “wrong” value have been detected.

## 8.7 Webdriver DomainsAllowed

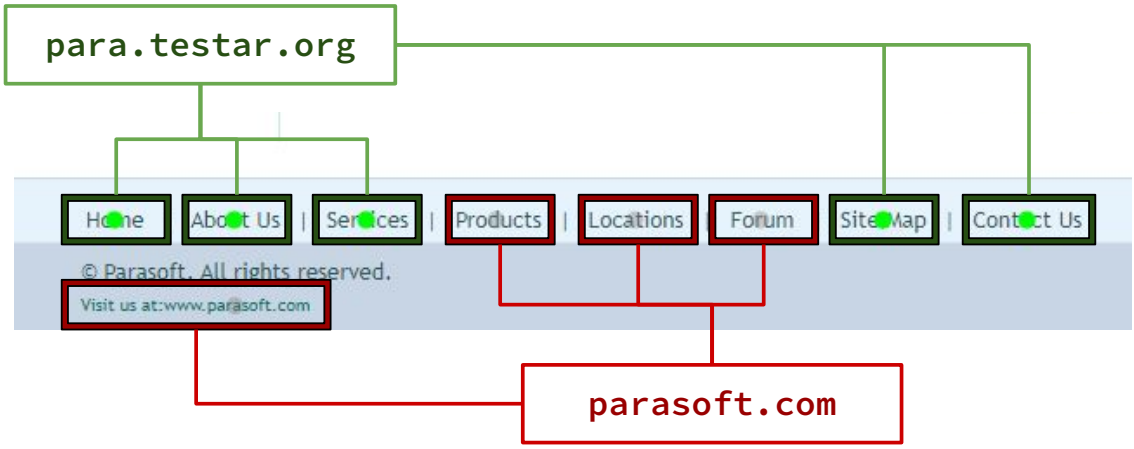
One of the features of using TESTAR with the webdriver, is the list of domains allowed represented as a `DomainsAllowed` setting. This feature is implemented to keep our tests in the same web domain since many web applications have links pointing to external domains.

Its purpose can be seen in the footer of the Parabank webpage. `Home`, `About Us`, `Services`, `Site Map`, and `Contact Us` widgets are hyperlinks to pages within the `para.testar.org` tested domain. These are all regarded as clickable by TESTAR and thus decorated with

the familiar green dot. Nonetheless, widgets such as `Products`, `Locations`, `Forum`, and `Visit us` are hyperlinks to the external domain `parasoft.com`.

Because `parasoft.com` is not included in the `DomainsAllowed` setting, we can observe a grey dot over the widget that indicates these widgets are potentially clickable but are filtered, in this case, due to web domain reasons. Adding the domain to the class setting `DomainsAllowed` allows these links to be visited.

```
################################################################
# WebDriver features
#
# ClickableClasses: Indicate which web CSS classes need to be considered clickable
# TypeableClasses: Indicate which web CSS classes need to be considered typeable
# DeniedExtensions: Indicate which web URL extensions need to be ignored when testing
# DomainsAllowed: Indicate which web URL domains need to be ignored when testing
# FollowLinks: Indicate if allowing to follow links opened in new tabs
# BrowserFullScreen: Indicate if perform the web testing with the browser in full screen
# SwitchNewTabs: Indicate if switch to a new web tab if opened
################################################################

ClickableClasses =
TypeableClasses =
DeniedExtensions = pdf;jpg;png;pfx;xml
DomainsAllowed = para.testar.org
FollowLinks = false
BrowserFullScreen = true
SwitchNewTabs = true
```




**hands-on 35**

Add the `parasoft.com` domain to the `DomainsAllowed` setting.

Open the test.settings file that exists in the parabank protocol folder:

`testar/bin/settings/02_webdriver_parabank/test.settings`

Find the `DomainsAllowed` setting, and add the `parasoft.com` domain separated by the semicolon `;` character

```
DomainsAllowed = para.testar.org;parasoft.com
```

Run the SPY mode and check that the widgets that contain hyperlinks to the `parasoft.com` domain are decorated with a green dot.

If we configure this `DomainsAllowed` setting as `null`, we are going to indicate to TESTAR that all web domains are allowed by default.

**hands-on 36**

Allow all web domains by declaring `DomainsAllowed` setting as `null`.

Open the test.settings file that exists in the parabank protocol folder:

`testar/bin/settings/02_webdriver_parabank/test.settings`

Find the `DomainsAllowed` setting, and add configure the variable as `null`

```
DomainsAllowed = null
```

Run the SPY mode and check all widgets that contain hyperlinks are decorated with a green dot.

Finally, TESTAR considers the domain of the URL to test must be added as an allowed domain by default. This means we do not need to specifically declare `para.testar.org` as a `DomainsAllowed` setting when testing the URL `https://para.testar.org/`.

**hands-on 37**

Default allowed web domains.

Open the test.settings file that exists in the parabank protocol folder:

`testar/bin/settings/02_webdriver_parabank/test.settings`

Find the `DomainsAllowed` setting and remove all values

```
DomainsAllowed =
```

Run the SPY mode and check that TESTAR automatically considers the widgets that contain hyperlinks to `para.testar.org` clickables with a green dot. But not the other widgets with hyperlinks that point out of the default web domain.

## 8.8 Webdriver DeniedExtensions

Some web pages may contain hyperlink widgets that point to images (png, jpg, etc.), documents, PDFs, or other files, that provoke downloading resources in the computer or that open new pages that do not have any interactive elements. Downloading these files or opening these non-interactive pages can cause TESTAR to get stuck and consume resources. For example, in `Parabank -> About Us` web page, we can find a clickable widget that downloads a `pfx` file in the system.


TESTAR implements a feature to deny executing actions on these undesired extensions. This feature can be customized using the `DeniedExtensions` setting.

**hands-on 38**

Remove `pfx` from the `DeniedExtensions` setting

Open the test.settings file that exists in the parabank protocol folder:

```
testar/bin/settings/02_webdriver_parabank/test.settings
```

Find the `DeniedExtensions` setting

```
DeniedExtensions = pdf;jpg;png;pfx;xml
```

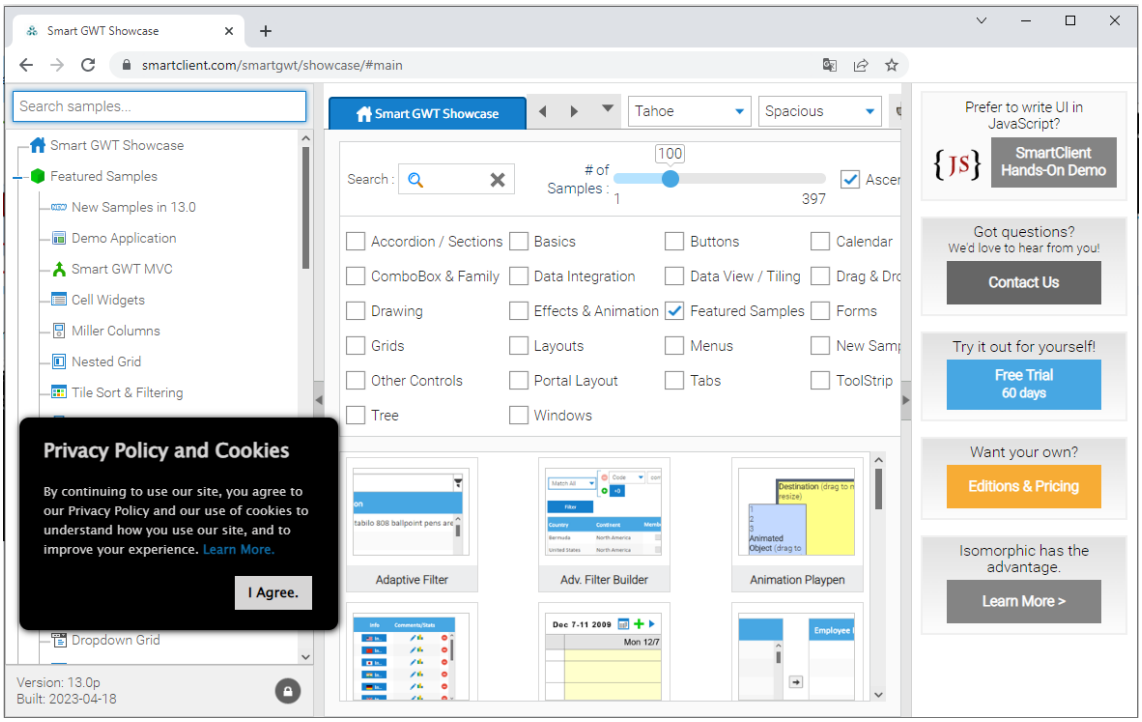
and remove the `pfx` value

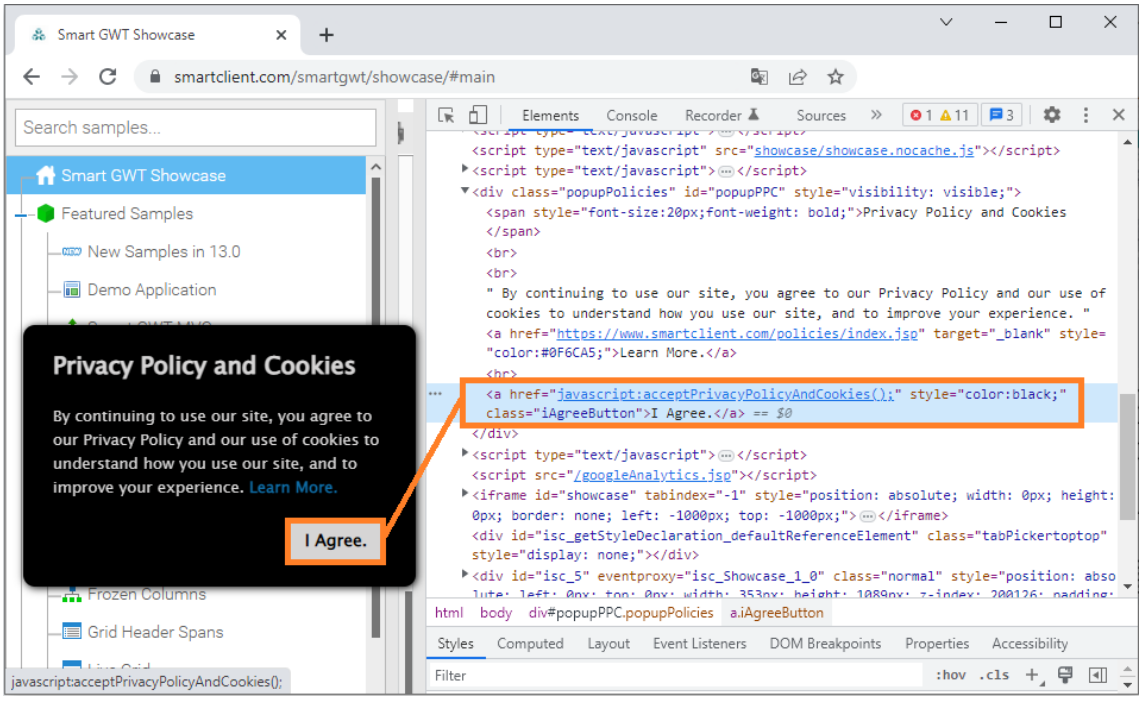
```
DeniedExtensions = pdf;jpg;png;xml
```

Run the SPY mode and change to `About Us` page to check that TESTAR considers actionable the `soatest.pfx` element when the `DeniedExtensions` setting is not configured to deny it. Finally, restore the `pfx` value again.

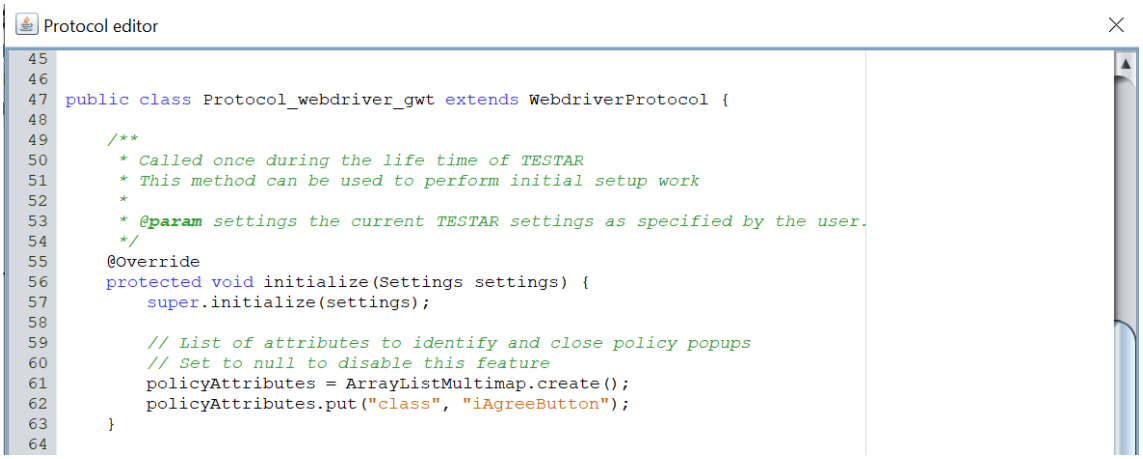
## 8.9 Policy and cookies panels

In the Parabank web application, we did not find this typical behavior. However, when you open a web application to interact, you will notice a familiar obstacle that may interfere with the testing process of TESTAR. Policy and cookies panels can be implemented to block the interaction with web elements until the user accepts or rejects the web policies. Therefore, TESTAR implements an additional feature that allows users to detect and handle them.



These policy elements are not always implemented with the same web attributes or using the same web elements identifiers. For this reason, TESTAR implements a feature that requires user customization in order to indicate which web attribute with which value represents the policy elements. This feature can be found in the `initialize()` method as a `policyAttributes` variable.




**hands-on 39**

Empty policy attributes feature

Open the General panel in the TESTAR dialog and change to the `webdriver_gwt` protocol. We are going to change the web application to test to

`https://www.smartclient.com/smartgwt/showcase/`


Edit the protocol, find the `initialize()` method, and comment out the line that maps the policy attribute with the agree button.

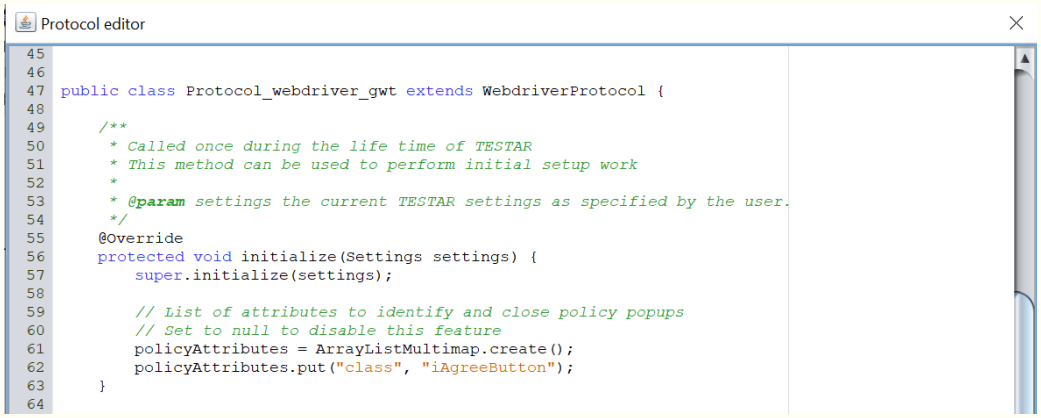
```
@Override
protected void initialize(Settings settings) {
    super.initialize(settings);

    // List of attributes to identify and close policy popups
    // Set to null to disable this feature
    policyAttributes = ArrayListMultimap.create();
    //policyAttributes.put("class", "iAgreeButton");
}
```

Run the SPY mode with TESTAR and notice all the clickable widgets (green dots) that are available when starting the web application.

**hands-on 40**

Custom policy attributes feature

In the same `webdriver_gwt` protocol. Edit the protocol, find the `initialize()` method, and enable again the line of code that maps the policy attribute with the agree button.




Run the SPY mode and notice that TESTAR only focuses on the `I Agree` widget button (green dot).

## 8.10 Webdriver clickable elements

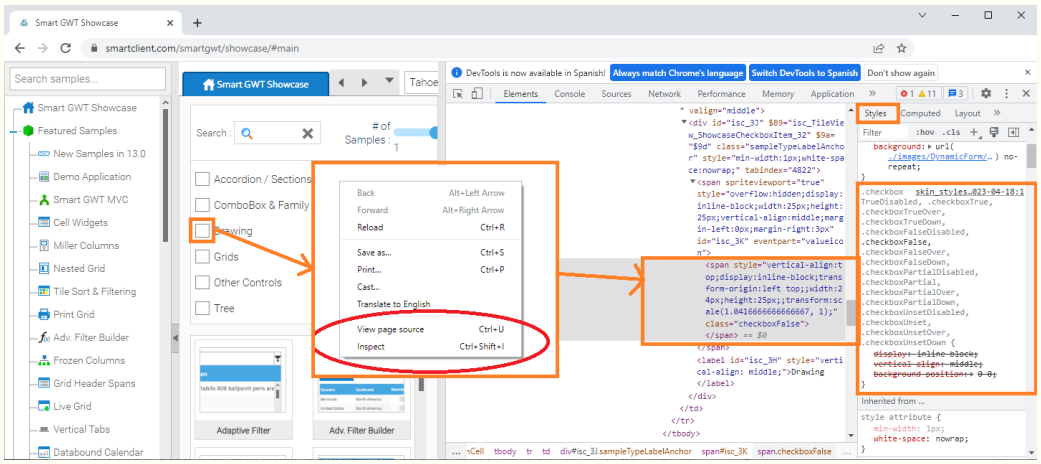
TESTAR considers, by default, that the clickable widgets of web applications are those that correspond to native web elements (hyperlinks, buttons, input submit, input radio, etc.). However, some web pages can contain non-native elements which are nevertheless clickable. The `ClickableClasses` setting allows users to indicate to TESTAR which non-native widgets must be considered clickable. In order to do that, we need to know the `class` web attribute [7] of the desired web element.

**hands-on 41**

Inspect the non-native clickable widgets

Open the `https://www.smartclient.com/smartgwt/showcase/` web page in a Chrome browser. With a Chrome browser, hover the mouse over a checkbox and use the mouse to right-click a checkbox and select Inspect.

[7] https://www.w3schools.com/tags/att_class.asp



The checkbox widgets are multiple $< span >$ web elements[a], mainly used to mark up a part of a text or a part of a document.

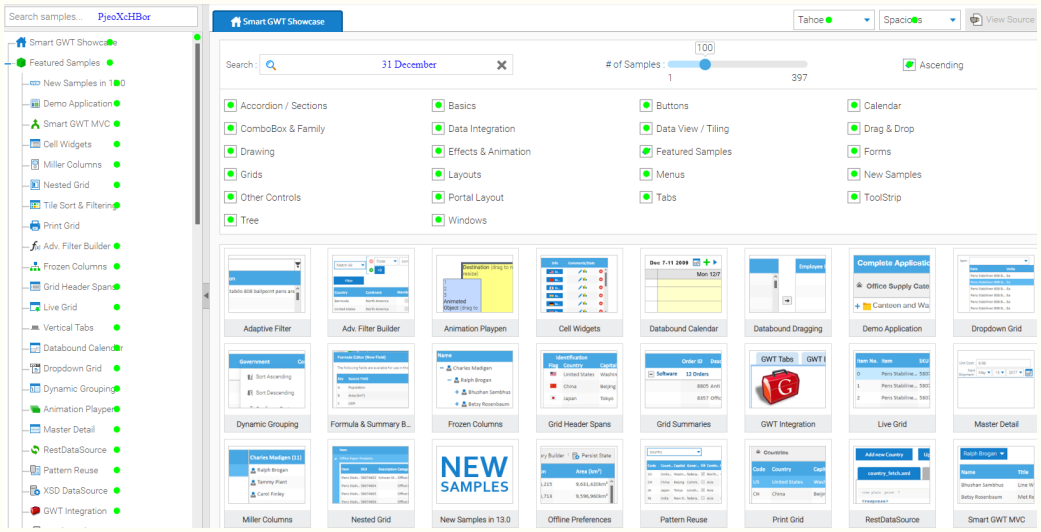
TESTAR can not consider all $< span >$ elements clickable by default because that will provoke deriving action in a lot of non-interactive widgets. But we can find the CSS classes of these $< span >$ elements to mark them in the `ClickableClasses` setting.

[a] https://www.w3schools.com/tags/tag_span.asp

## hands-on 42

Custom the `DomainsAllowed` setting in TESTAR

① Run TESTAR and select the `webdriver_gwt` protocol. Launch the SPY mode, manually agree with the policy button, and verify TESTAR derives clickable actions (green dots) in all the checkboxes.



② Stop TESTAR or close the web application. Open the test.settings file that exists in the parabank protocol folder:

`testar/bin/settings/02_webdriver_parabank/test.settings`

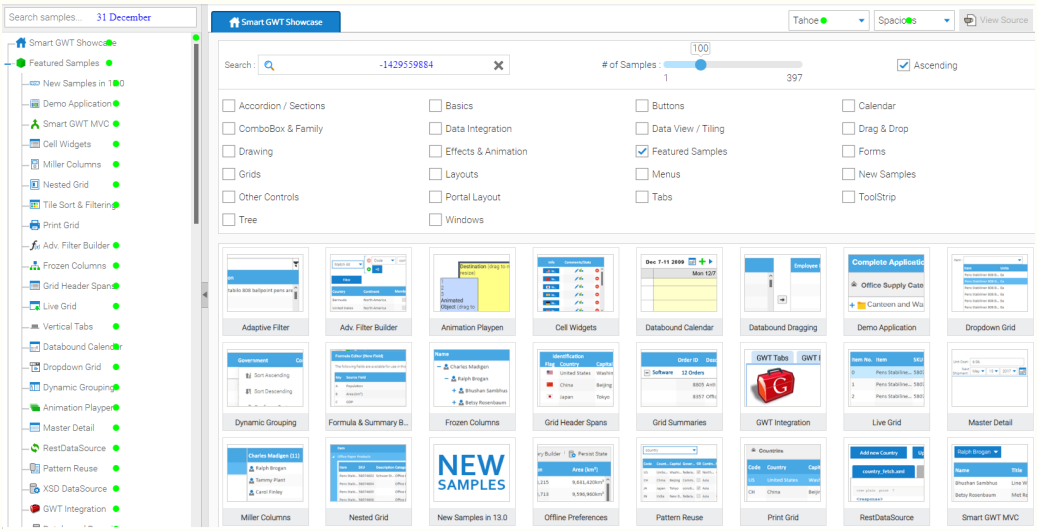
Find the `DomainsAllowed` setting

ClickableClasses = selectItemLiteText;etreeCell;etreeCellSelected;
etreeCellSelectedOver;checkboxFalse;checkboxFalseOver;checkboxTrue;
checkboxTrueOver;showcaseTileIcon;vScrollStart;vScrollEnd

and remove all the `checkboxFalse` and `checkboxTrue` classes values.

ClickableClasses = selectItemLiteText;etreeCell;etreeCellSelected;
etreeCellSelectedOver;showcaseTileIcon;vScrollStart;vScrollEnd

③ Run TESTAR with the `webdriver_gwt` protocol again. Launch the SPY mode, manually agree with the policy button, and verify TESTAR does not derive clickable actions (green dots) in the checkboxes.




④ Stop TESTAR or close the web application. Open the test.settings file that exists in the parabank protocol folder:

`testar/bin/settings/02_webdriver_parabank/test.settings`

Find the `DomainsAllowed` setting and add the `thumbnail` class value

ClickableClasses = selectItemLiteText;etreeCell;etreeCellSelected;
etreeCellSelectedOver;showcaseTileIcon;vScrollStart;vScrollEnd;thumbnail

④ Run TESTAR with the `webdriver_gwt` protocol again. Launch the SPY mode and manually agree with the policy button. You will notice the new thumbnail clickable widgets (green dots).

SECTION 9

# Advanced TESTAR Oracles

We have seen the TESTAR capabilities and required configurations to detect if the SUT:

1. closes unexpectedly by a crash
2. stops responding due to a freeze
3. GUI widgets of Desktop and Web apps, the output buffer of Desktop apps, or the browser console of Web apps contains suspicious Tags messages.

TESTAR protocols can be extended programmatically to go further in the detection of test oracles. Sometimes, even if the SUT remains robust without throwing errors, we can detect other types of malfunctions in the applications. For example, SUT tables with empty or duplicated rows or columns, selection elements without items or duplicated values, or text inputs on which it is not possible to type due to internal code bugs.

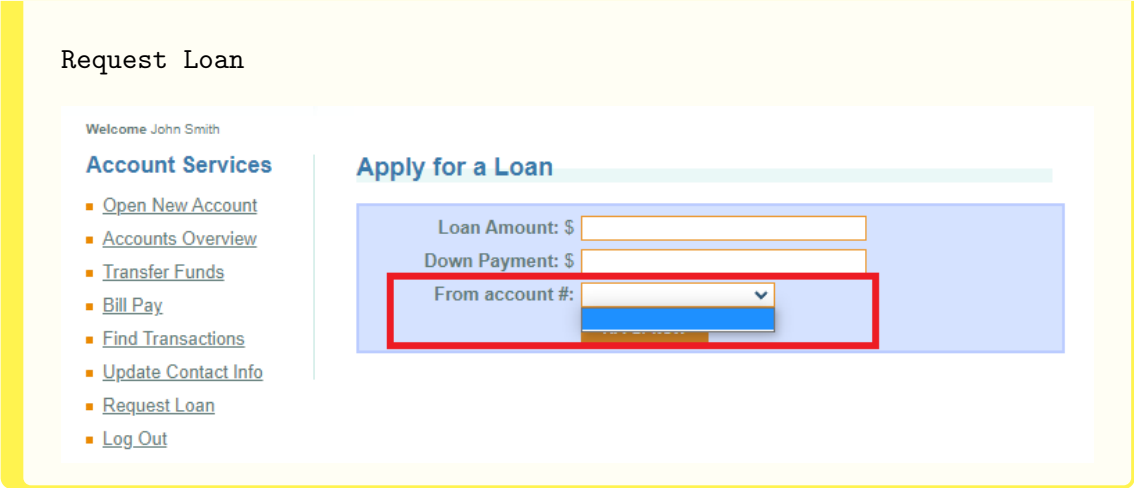
**hands-on 43**

Manually connect to Parabank to check for logical programming malfunctions

Open the URL `https://para.testar.org/` in your browser and log in with username=john and password=demo credentials.

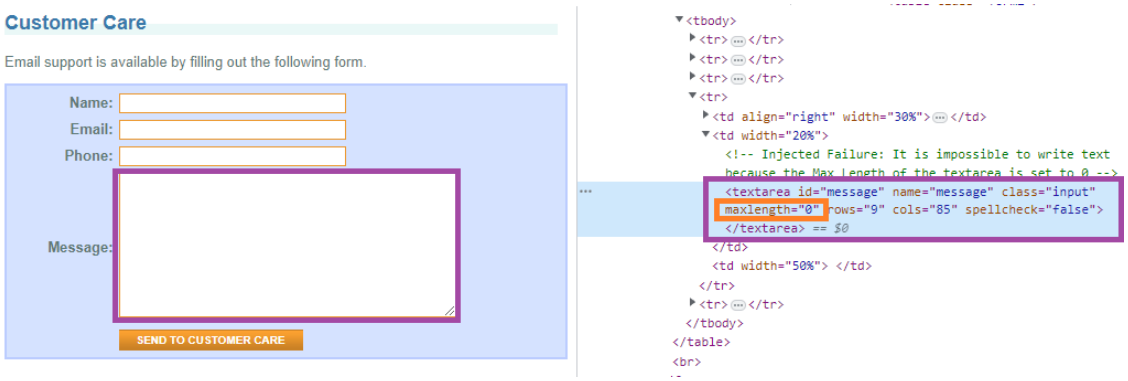
Check at least the next web pages for malfunctions:

`Accounts Overview`

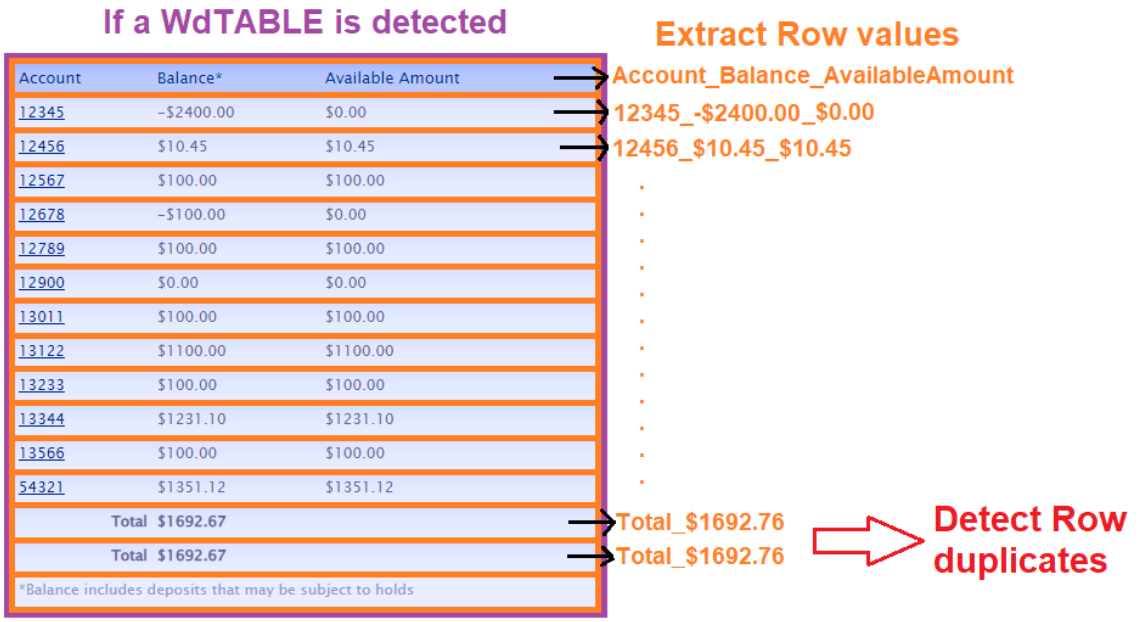



These bugs can be detected by extending the `getVerdict()` method and programming customized methods that iterate through the state of TESTAR by checking specific widgets and their properties, or executing queries using the WebDriver execute script feature `WdDriver.executeScript(query)`

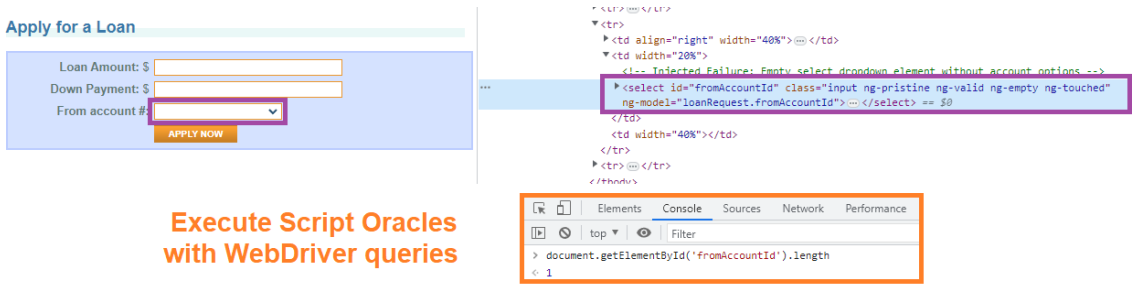
This first example represents a malfunction that only requires checking one property of all the widgets of the web state in order to detect that the `maxlength` property of a `textArea` widget was wrongly programmed with a 0 value.




The second example represents a malfunction that creates tables with duplicated rows. This can be more complicated to customize in terms of protocol programming. In the states we detect the existence of a web table element, we can extract the values of all existing rows to apply a function that searches for duplicates.




Finally, the next example that represents a malfunction with only one empty item in a select dropdown web element can be detected using the WebDriver execute script feature. For each select widget of the state, we can extract the web identifier using the TESTAR `WebId` property, then execute a query that checks the length value of the select widget.




**hands-on 44**

Run TESTAR to detect Parabank malfunctions

① Run TESTAR and select the `02_webdriver_parabank` protocol. Then, click the `Edit Protocol` button to open the Java protocol.

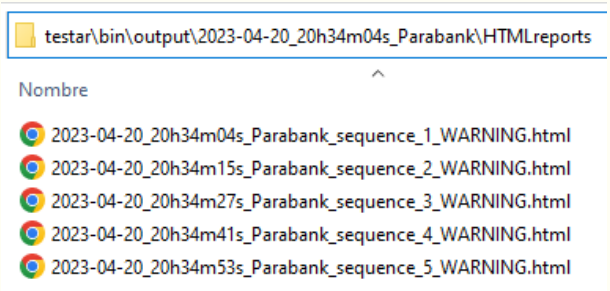
② Find the `getVerdict()` method and uncomment the custom verdicts that invoke the examples of advanced oracles methods. Then, Save and Compile the protocol.

Protocol editor

```
153
154     /**
155      * This is a helper method used by the default implementation of <code>buildState()</code>
156      * It examines the SUT's current state and returns an oracle verdict.
157      *
158      * @return oracle verdict, which determines whether the state is erroneous and why.
159      */
160     @Override
161     protected Verdict getVerdict(State state) {
162
163         // System crashes, non-responsiveness and suspicious tags automatically detected!
164         // For web applications, web browser errors and warnings can also be enabled via settings
165         Verdict verdict = super.getVerdict(state);
166
167         // If the Verdict is not OK but was already detected in a previous sequence
168         // Consider as OK to avoid duplicates and continue testing
169         if (verdict != Verdict.OK && containsVerdictInfo(listOfDetectedErroneousVerdicts, verdict.info())) {
170             // Consider as OK to continue testing
171             verdict = Verdict.OK;
172             webConsoleVerdict = Verdict.OK;
173         }
174         // If the Verdict is not OK and was not duplicated...
175         // We found an issue we need to report
176         else if (verdict.severity() != Verdict.OK.severity()) {
177             return verdict;
178         }
179
180         //---------------------------------------------------------------------------
181         // MORE SOPHISTICATED ORACLES CAN BE PROGRAMMED HERE (the sky is the limit ;-)
182         //---------------------------------------------------------------------------
183
184         // ... YOU MAY WANT TO CHECK YOUR CUSTOM ORACLES HERE ...
185
186         Verdict customVerdict = Verdict.OK;
187         /*
188         customVerdict = customVerdict.join(detectNumberWithLotOfDecimals(state, 2));
189
190         customVerdict = customVerdict.join(detectEmptySelectItems(state));
191
192         customVerdict = customVerdict.join(detectDuplicateSelectItems(state));
193
194         customVerdict = customVerdict.join(detectTextAreaWithoutLength(state, Arrays.asList(WdRoles.WdTEXTAREA)));
195
196         customVerdict = customVerdict.join(detectDuplicatedRowsInTable(state));
197
198         // If the Custom Verdict is not OK but was already detected in a previous sequence
199         // Consider as OK to avoid duplicates
200         if (customVerdict != Verdict.OK && containsVerdictInfo(listOfDetectedErroneousVerdicts, customVerdict.info())) {
201             customVerdict = Verdict.OK;
202         }
203          */
204         return customVerdict;
205     }
206
```

Save and Compile

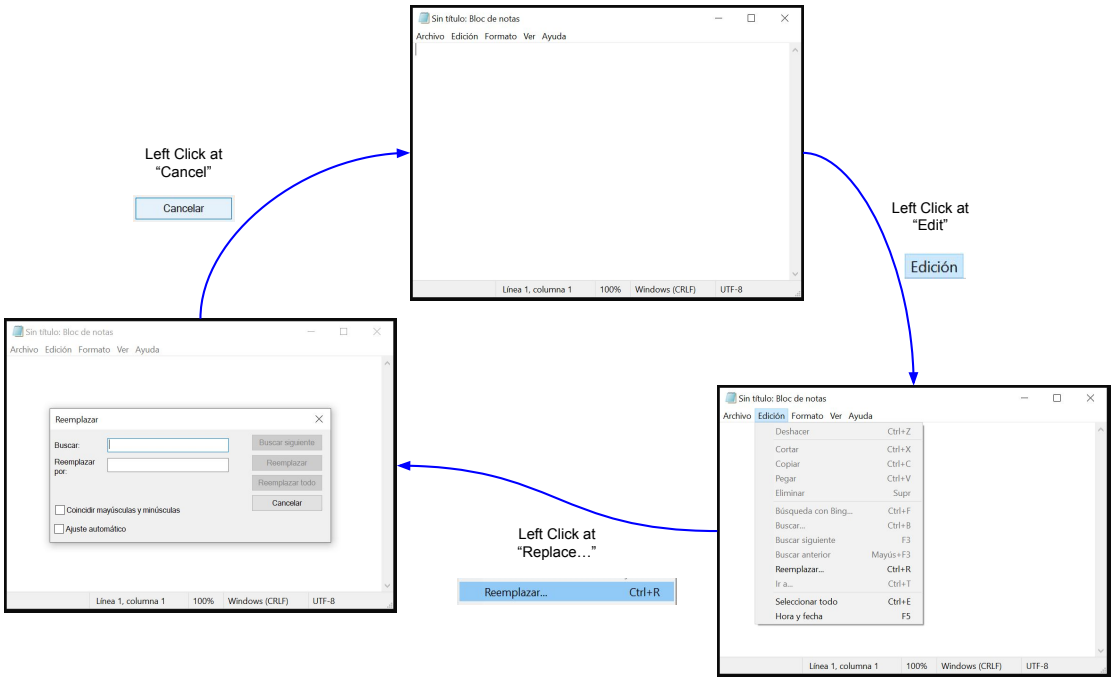
③ Run the Generate mode of TESTAR. After some sequences, check in the HTMLreports if TESTAR was able to detect some `WARNING` sequences



SECTION 10

# TESTAR State Model

As TESTAR explores and tests the SUT, it can generate a State Model that traces which states were discovered while executing GUI actions. This State Model is stored in the OrientDB graph database. After executing test sequences and generating the State Model, users can query the database using OrientDB studio or analyze the model using TESTAR Analysis mode.




## 10.1 Install OrientDB

`Warning:` Current OrientDB version is `3.2.X`. However TESTAR currently requires the use of versions `3.0.X`

You can find the OrientDB Community Edition versions to download from:
`https://repo1.maven.org/maven2/com/orientechnologies/orientdb-community/`

You will need to create a database with user and password credentials. Next `Option 1 subsection` indicates a URL to download an already customized database. And `Option 2 subsection` provides manual instructions to do this configuration.

### 10.1.1 Option 1: Use a configured TESTAR OrientDB

The TESTAR team already provides a distribution of configured OrientDB software. This distribution contains a database with the following `testar` configuration:

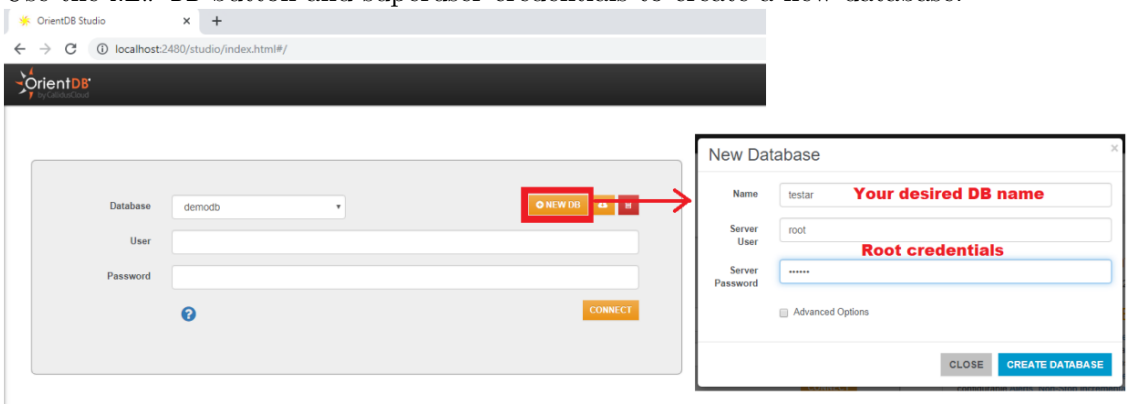
- database name: `testar`
- username: `testar`
- password: `testar`

`https://testar.org/images/development/experiments/orientdb_testar_db.zip`

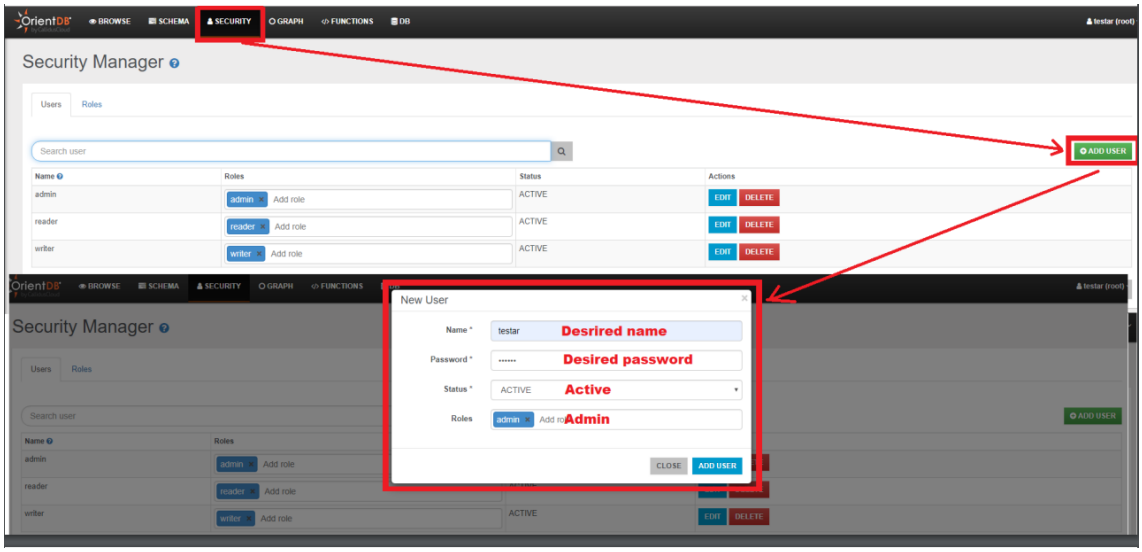
### 10.1.2 Option 2: Manual configuration of OrientDB

1. Extract all OrientDB files in the desired directory.
2. Execute:
   `orientdb-3.0.X\bin\server.bat` for Windows hosts
   `orientdb-3.0.X\bin\server.sh` for Unix hosts
3. OrientDB server will start running on the system. In the first OrientDB execution, you need to create superuser credentials in the opened command prompt.


4. Use your browser to access to `http://localhost:2480/`
5. Use the `NEW DB` button and superuser credentials to create a new database.




6. Click on `Security` tab and `ADD USER` button to create a new user with access to the database. For basic experimentation we recommend to use `name:testar password:testar` Use `Status:Active` and `Roles:admin` for permissions.

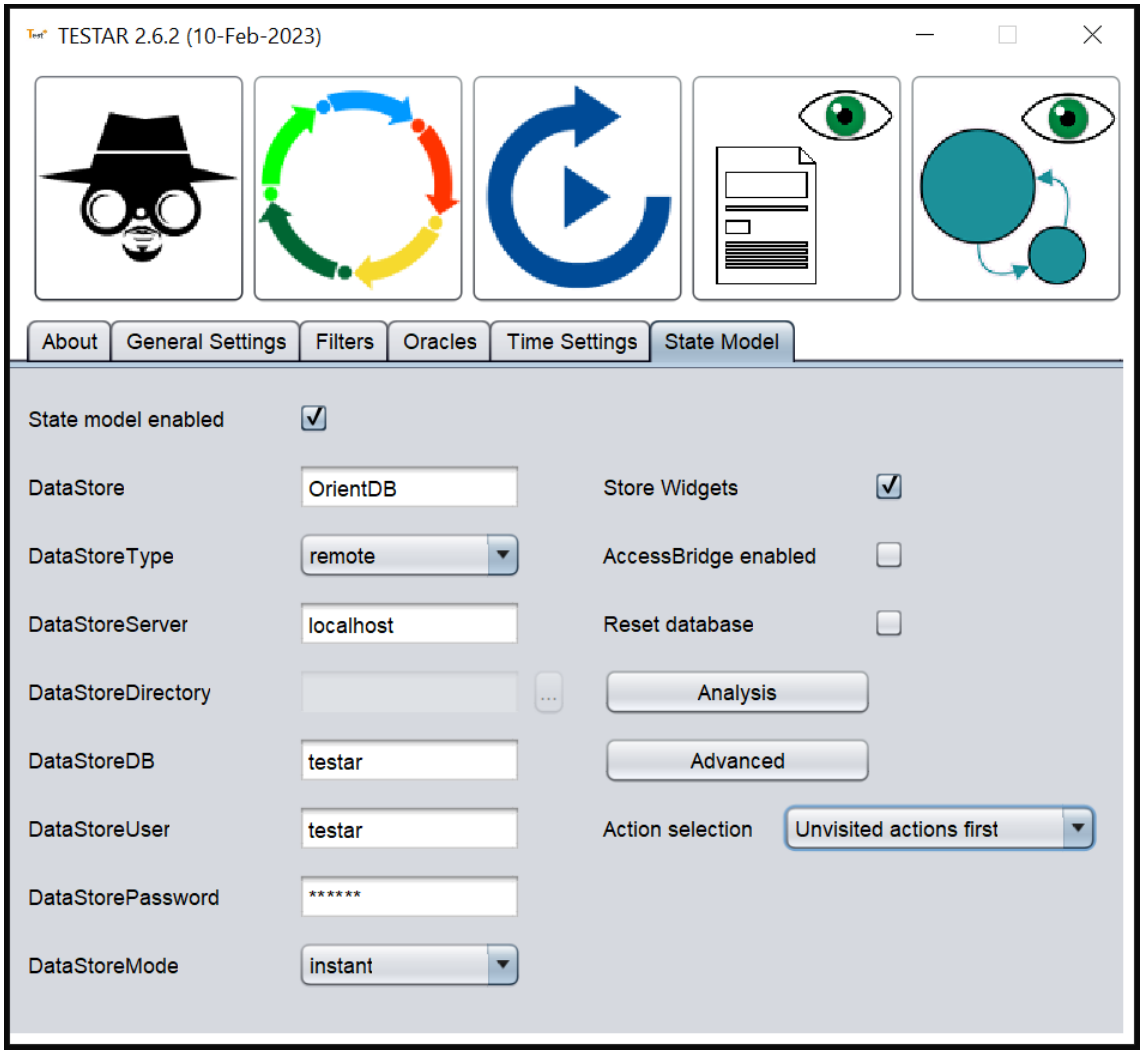



After these configuration steps, you can move to the running OrientDB command prompt and press `Control + C` to stop the execution.

## 10.2 Configure TESTAR State Model settings

We need to configure multiple settings in TESTAR to indicate to the tool how to connect with the OrientDB server and which features to use. These settings can be configured using the State Model panel in the TESTAR dialog or by editing the test.settings file.

The `State model enabled` checkbox, which corresponds to the `StateModelEnabled` setting, indicates whether TESTAR is connected to OrientDB or not.



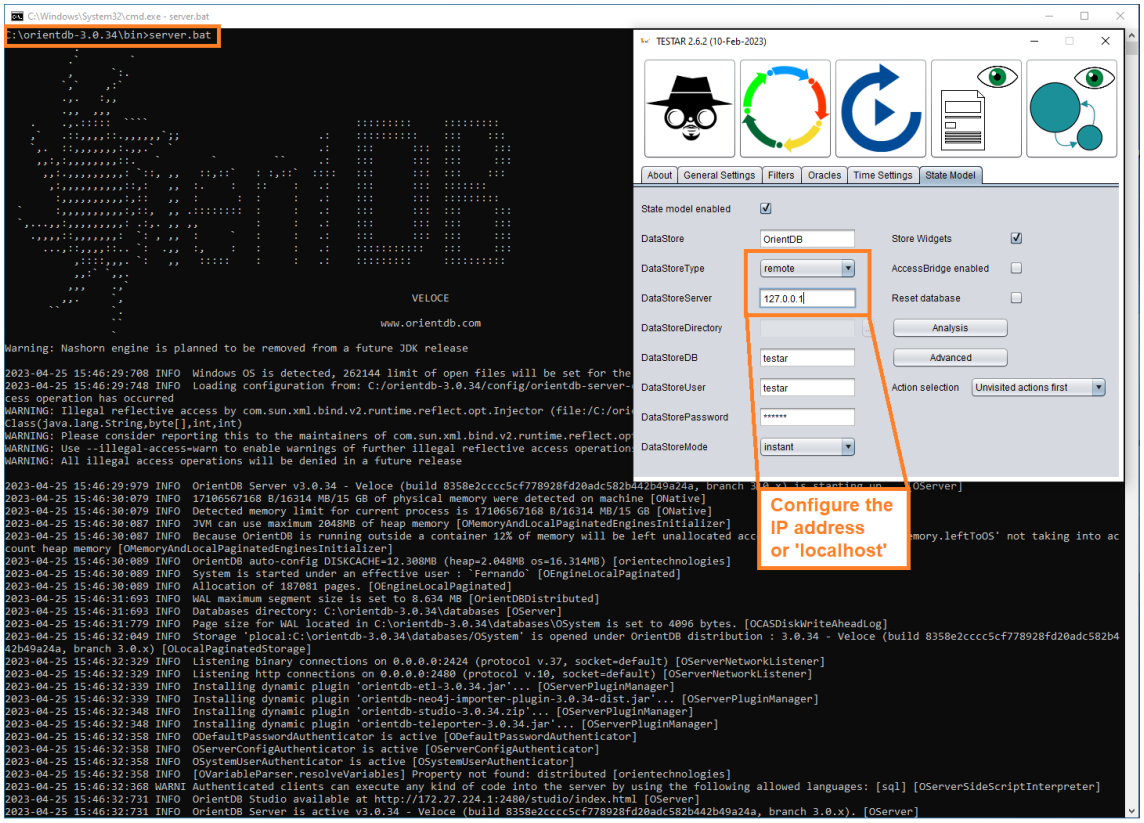

test.settings

```
######################################################################
# State model inference settings
#
# StateModelEnabled: Enable or disable the State Model feature
# DataStore: The graph database we use to store the State Model: OrientDB
# DataStoreType: The mode we use to connect to the database: remote or plocal
# DataStoreServer: IP address to connect if we desired to use remote mode
# DataStoreDirectory: Path of the directory on which local OrientDB exists, if we use plocal mode
# DataStoreDB: The name of the desired database on which we want to store the State Model.
# DataStoreUser: User credential to authenticate TESTAR in OrientDB
# DataStorePassword: Password credential to authenticate TESTAR in OrientDB
# DataStoreMode: Indicate how TESTAR should store the model objects in the database: instant, delayed, hybrid, none
# ApplicationName: Name to identify the SUT. Especially important to identify a State Model
# ApplicationVersion: Version to identify the SUT. Especially important to identify a State Model
# ActionSelectionAlgorithm: State Model Action Selection mechanism to explore the SUT: random or unvisited
# StateModelStoreWidgets: Save all widget tree information in the State Model every time TESTAR discovers a new Concrete State
# ResetDataStore: WARNING: Delete all existing State Models from the selected database before creating a new one
######################################################################

StateModelEnabled = true
DataStore = OrientDB
DataStoreType = remote
DataStoreServer = localhost
DataStoreDirectory =
DataStoreDB = testar
DataStoreUser = testar
DataStorePassword = testar
DataStoreMode = instant
ApplicationName =
ApplicationVersion =
ActionSelectionAlgorithm = unvisited
StateModelStoreWidgets = true
ResetDataStore = false
```

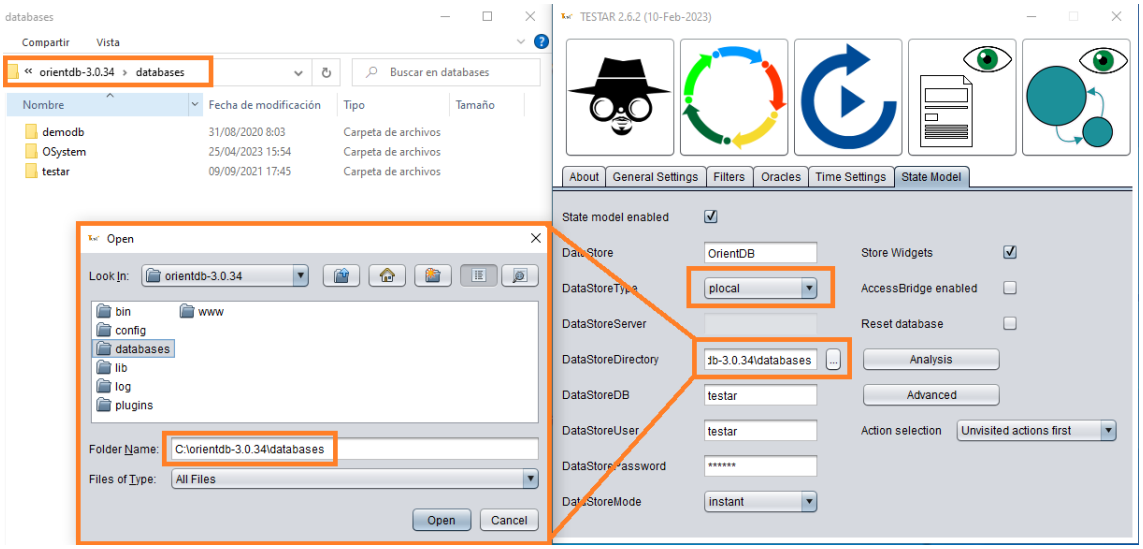
### 10.2.1 OrientDB connection mode

We have two ways to connect TESTAR with OrientDB database: `remote` and `plocal`

`remote` mode uses a network connection that allows us to connect to an IP address. If we have run the OrientDB server (server.bat), what we have done has been to run the server on our local machine, so we can use the localhost address to connect to the server that runs on our own machine.




`plocal` mode connects to the database at the file level, reading and writing directly to the disk. To enable this mode, we need to indicate in `DataStoreDirectory` setting the path to the directory of the `orientdb-3.0.X\databases`.




For both modes, we will need to indicate the previously configured OrientDB settings:

- `DataStoreDB` that corresponds with the name of the database.
- `DataStoreUser` that corresponds with the username.
- `DataStorePassword` that corresponds with the password.

### 10.2.2 Other State Model settings

`DataStoreMode` parameter must remain with the OrientDB value

`DataStoreMode` parameter is used to indicate how TESTAR should store the graphic objects (vertices and edges) in the database:

1. `Instant` : All data will be stored after each action. *This is the default and recommended mode.*
2. `Delayed` : The data will remain in memory and stored at the end of the sequence.
3. `Hybrid` : Abstract data will be stored in Instant mode, and concrete data in Delayed mode.
4. `None` : No new data will be stored, only to read the State Model.

`Store Widgets` : is a feature that allows us to store the entire widget tree each time we discover a new concrete state.

Warning: For medium complexity applications, where it is common to find a lot of new Concrete States composed of many widgets in the GUI, this mode increases (a lot) the time that TESTAR takes to store the information in the database.

`Access Bridge Enabled` : feature to test Java Swing apps.

`Reset Database` : Delete all State Models from a database before creating the new model. *Warning: Be careful.*

`Unvisited Action Selection` : It uses the State Model information to navigate and prioritize to select discovered but not executed actions. The objective is trying to complete

the state model.

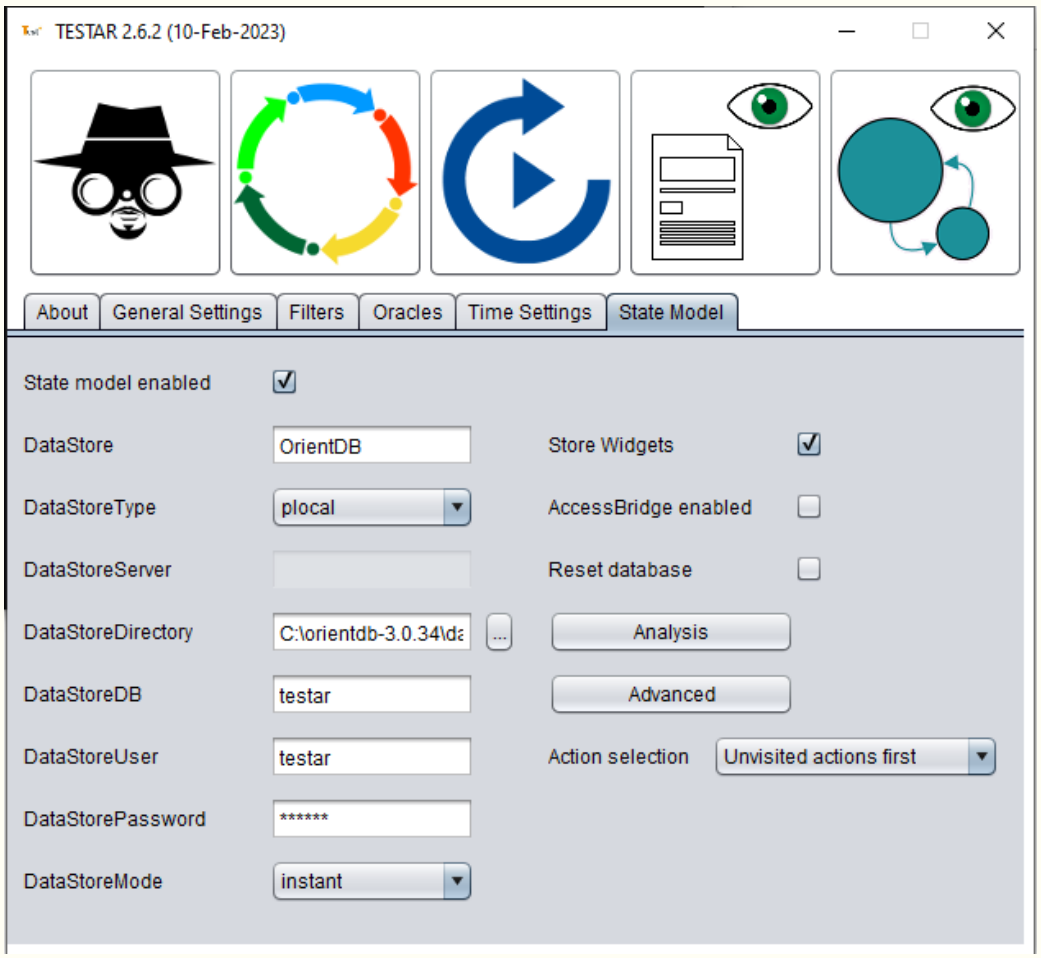
**hands-on 45**

Infer your first State Model with TESTAR

① Check that no OrientDB instances are running in your system (command prompt running server.bat).

② Run TESTAR, select the `desktop_generic` protocol, and change to the State Model panel.

③ Customize the dialog settings to connect to OrientDB using the `plocal` mode. Change the database name, user, and password if necessary. Enable `Store Widgets` option, disable `Access Bridge Enabled` and `Reset Database`, and select the `Unvisited actions first` algorithm.




④ Run 1 sequence of 10 actions with the Generate mode of TESTAR.

## 10.3 State Model Analysis

To view and analyze the generated State Model, we have two main options, use TESTAR to load the existing models and use a web browser, or access OrientDB Studio [8] to launch queries over the State Model objects, also using a web browser.

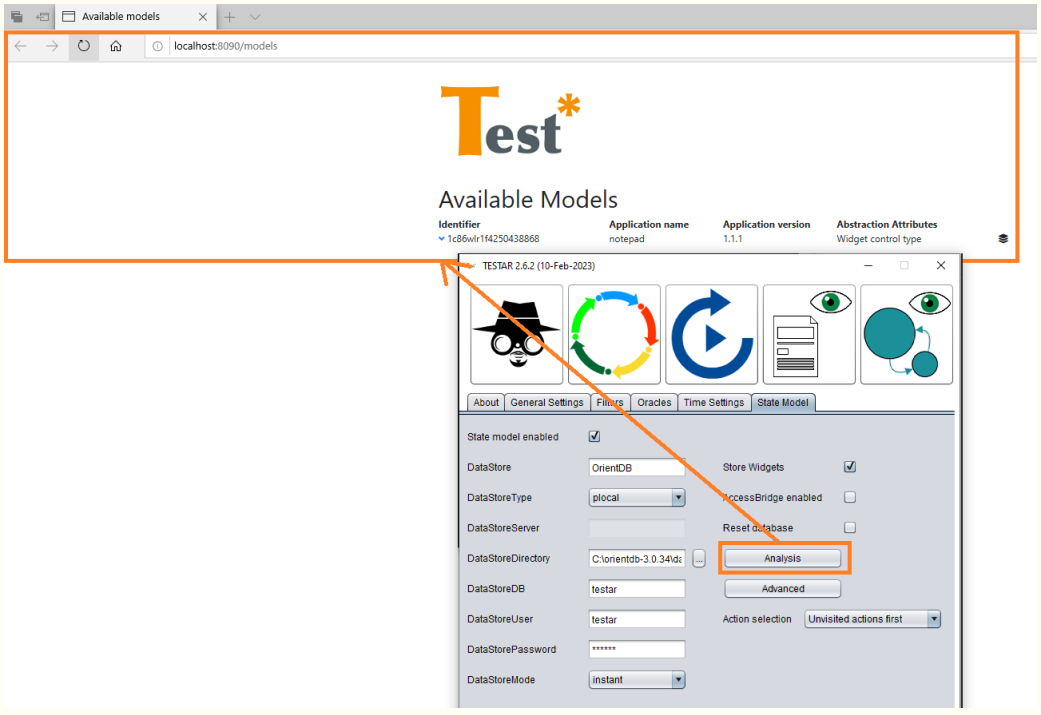
OrientDB Studio allows the user to perform post-analysis of TESTAR executions and execute queries to implement Offline Oracle Verdicts. However, in this Hands-On tutorial, we will focus on the TESTAR Analysis mode, which allows the user to visualize the navigational map (State-Action) that TESTAR found by exploring and testing the SUT. Additionally, users can click and interact with existing States and Actions of different layers to check the set of detected properties.

**hands-on 46**

Analyze your first TESTAR State Model

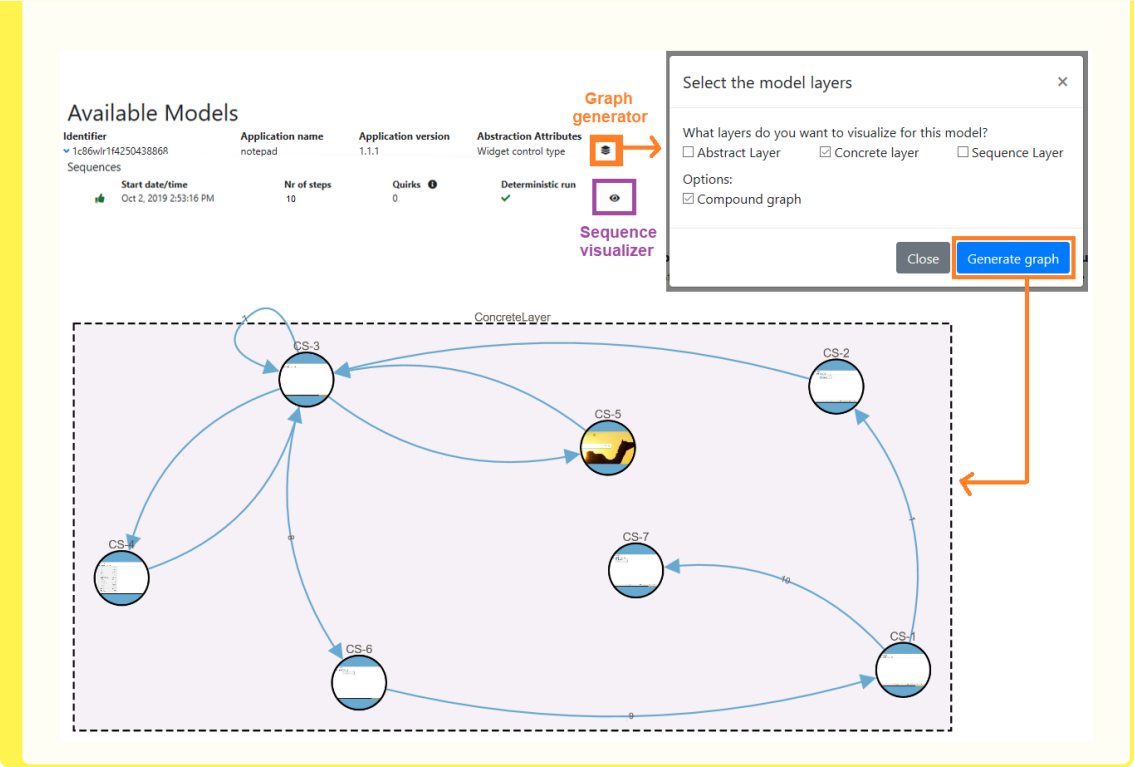
① Run TESTAR, select the `desktop_generic` protocol, and change to the State Model panel.

② If you changed some State Model settings, prepare the `plocal` mode, database name, user, and password settings again. Then, click the `Analysis` button, which opens the default system browser.




③ Click on the `Graph generator` web element, select the `Concrete layer` and the `Compound graph` options, and generate the visualization of the TESTAR State Model graph.

[8] http://orientdb.com/docs/3.0.x/studio/




As you can see in the TESTAR Analysis web, there are additional options, such as clicking on the `Sequence visualizer` web element to see similar step-by-step information like the HTML report. When generating the State Model graph, you can also build the `Abstract layer` and `Sequence layer`. Nonetheless, these other two layers are most important for the `Unvisited` algorithm that for visualization purposes.

## 10.4 State Model Abstraction

To check if we have found a new State in the SUT, TESTAR uses the set of properties of all the widgets in the GUI. This set of properties is checked after the execution of each action and generates two types of States:

1. `Concrete State` : ALL properties of ALL widgets are used. If any property appears, disappears, or changes, we have a new Concrete State.
2. `Abstract State` : SPECIFIC properties of ALL widgets are used. If one of these specific properties appears, disappears, or changes, we have a new Abstract State.

It's important to understand and configure this level of abstraction correctly to create a TESTAR State Model that accurately represents how States and Actions flow within an application.

On the one hand, working with very concrete states is not a good idea because any property change would constantly generate new States in the model. And on the other hand, too much abstraction can generate an incorrect State Model, where two different States are considered the same.

Given the following Figure 21, let's consider only the Widget Control Type (Role of the widget) to define the `Abstract State`. `Format` "Menu" is composed of two "Menu Items"

and `View` "Menu" is also composed of two "Menu Items". If we only consider the Widget Control Type (Role of the widget), these totally different States will be considered the same `Abstract State`. This provokes undesired behavior in the State Model algorithms because TESTAR is not able to calculate how to navigate to execute the unvisited actions properly.

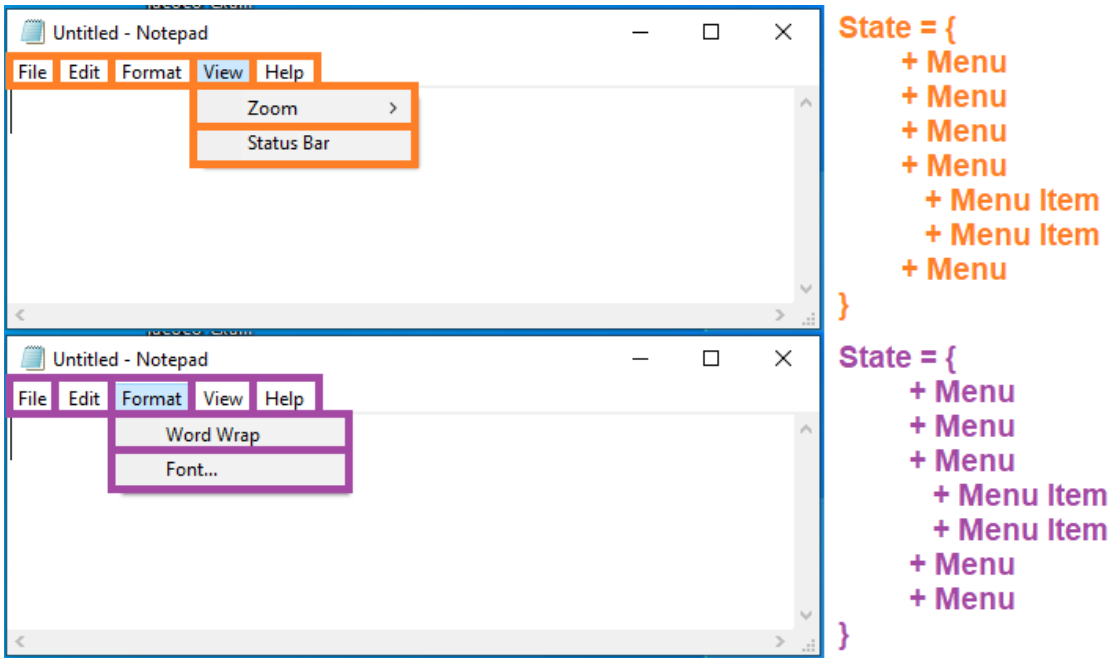



Figure 21: State Model Widget Control Type Abstraction

To solve this, in addition to using the widget control type, we can also use the widget title or the path in the widget tree as an abstract property.

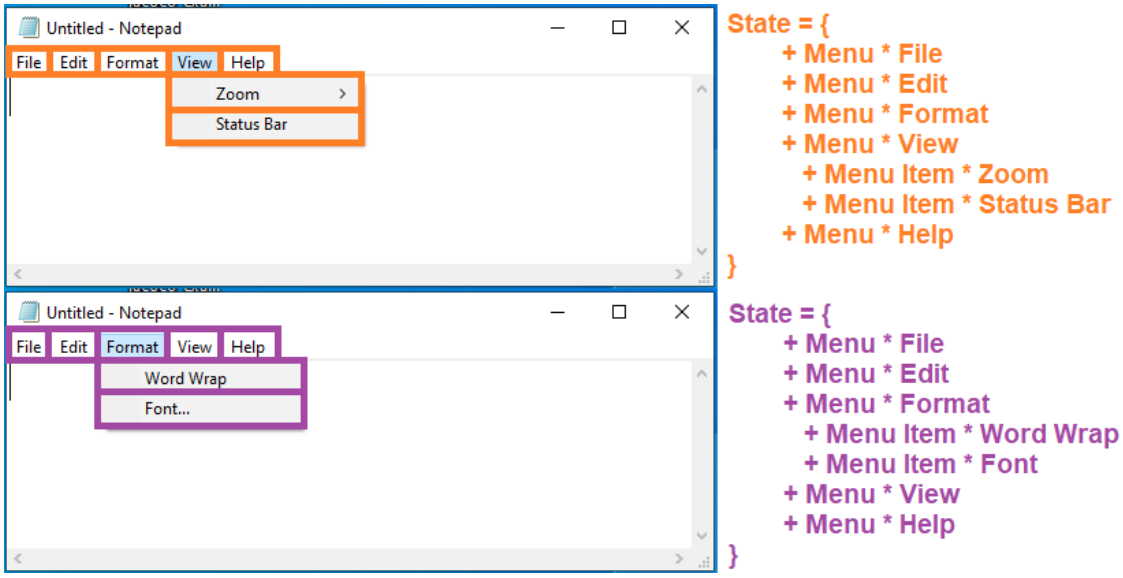
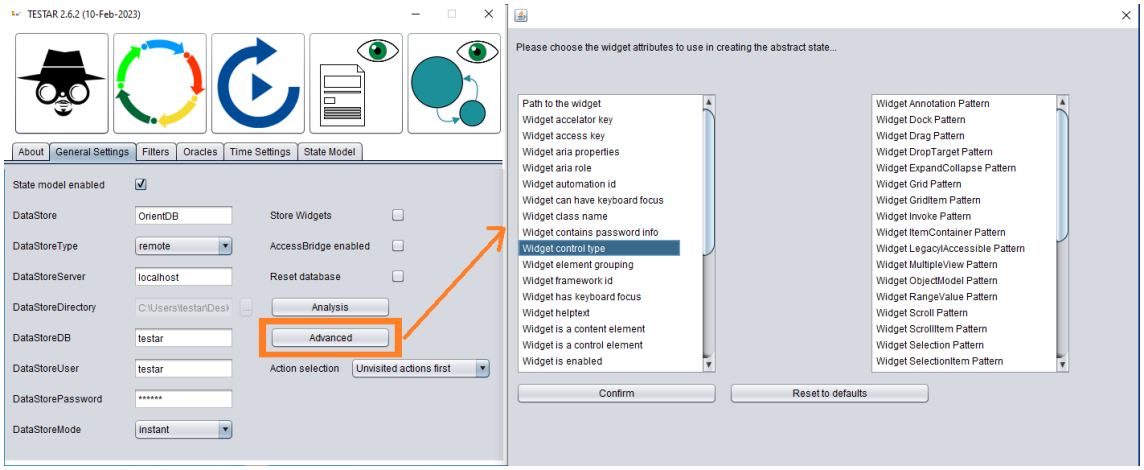



Figure 22: State Model Widget Control Type and Title Abstraction

### 10.4.1 State Model Advanced setting

The properties to customize the Abstraction layer of TESTAR can be modified through the test.settings file or using the `Advanced` button in the State Model panel of the TESTAR dialog.

In the State Model panel, use Control + Left Click to select or unselect the desired properties.




In the test.settings file, separate the desired properties by using commas.

test.settings

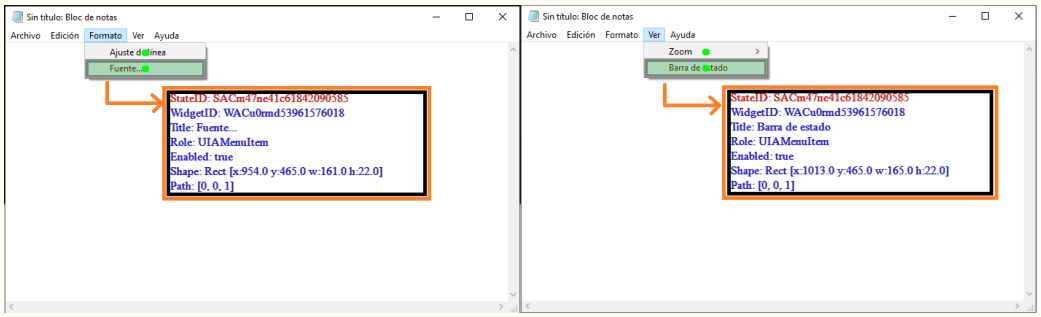
```
163  ######################################################################
164  # State identifier attributes
165  #
166  # Specify the widget attributes that you wish to use in constructing
167  # the widget and state hash strings. Use a comma separated list.
168  ######################################################################
169
170  AbstractStateAttributes = WidgetControlType,WidgetTitle
171
```

**hands-on 47**

Spy Notepad with different Abstraction properties

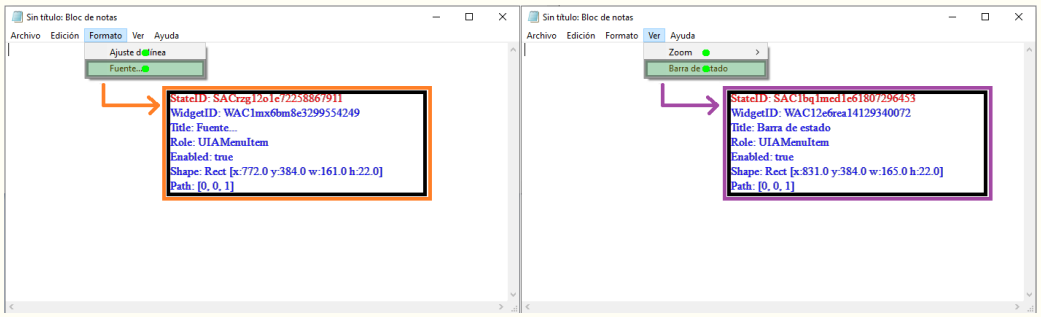
① Launch TESTAR, select the `desktop_generic` protocol, change to the State Model panel, open the `Advanced` Abstraction panel, and only select the `Widget Control Type`.

② Run the SPY mode with TESTAR, and Spy the `Format` and `View` menus to check the widgets and the state have the same abstract identifier.




③ Stop the SPY mode, and this time select both `Widget Control Type` and `Widget Title` properties.

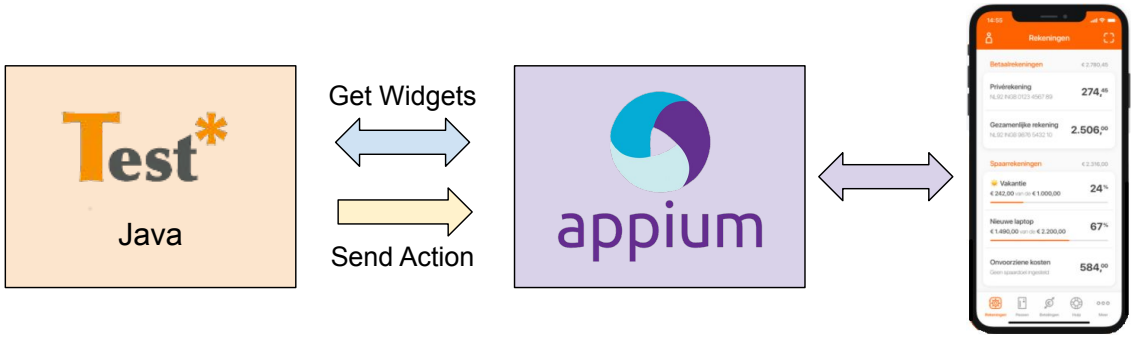
④ Run the SPY mode with TESTAR, and Spy the `Format` and `View` menus to check the widgets and the state now have different abstract identifiers.



SECTION 11

# Android systems

Appium[9] is an open-source test automation framework that allows extracting the information of mobile system elements and sending commands to interact with them. TESTAR integrates the java-client plugin of Appium in order to implement the AndroidDriver to test Android mobile systems and the IOSDriver to test iOS mobile systems. The next Figure shows how Appium works as a middleware between TESTAR and a mobile emulator/device to extract the widget's information and send actions.

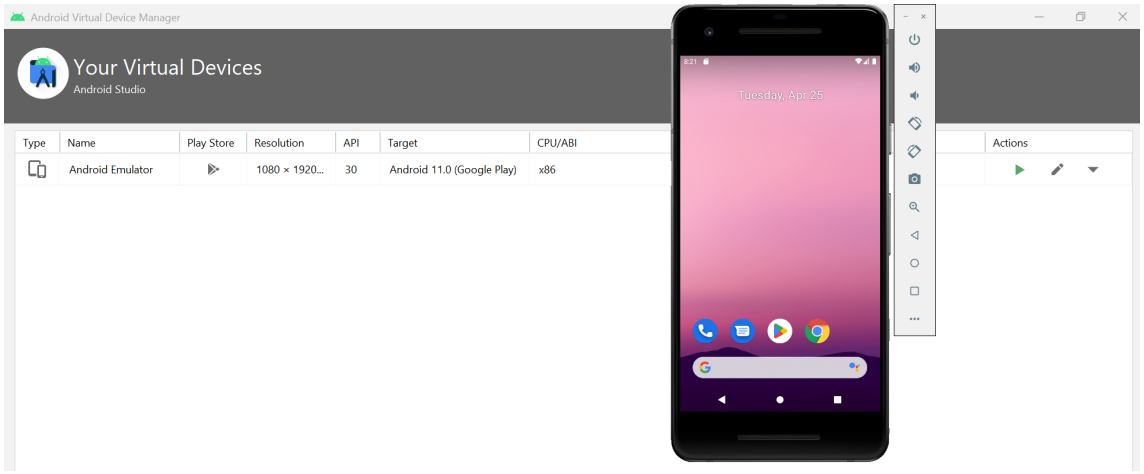



## 11.1 Preparing a mobile environment

Unlike Desktop and Web applications, for mobile applications, it is necessary to prepare a mobile environment that allows us to run the desired system under test. Users can connect a real mobile device to a computer or prepare different Android or iOS virtual emulators. In this guide, we recommend starting with the official Android Studio IDE[10]. Later, we will also describe the possibility of dockerizing the emulator and its software dependencies.

Android Studio contains the Android Virtual Device (AVD) feature, which allows specifying the Android version and hardware characteristics to launch a simulated device[11]. Once you have configured an AVD, you will be able to run a simulated mobile on your host computer. For the future TESTAR steps, you need to take note of the "AVD name" provided to the virtual mobile.

[9] https://appium.io/docs/en/2.0/
[10] https://developer.android.com/studio/run/emulator
[11] https://developer.android.com/studio/run/managing-avds




## 11.2 Installing Appium

Appium Desktop[12] was an old application that allowed to install a GUI desktop version of Appium to work as a server to connect with the emulator and execute Android test sequences. However, it is a deprecated project that has remained archived without maintenance since 2023. This section will indicate which software and instructions can help you install Appium on your computer using the `npm` package manager. You can find more documentation, details, and troubleshooting on the official Appium webpage [13]

1. **Install NodeJS and NPM:** NodeJS is a run-time environment software that allows to execution of programs written in JavaScript. And Node Package Manager (NPM) is an application and repository for developing and sharing JavaScript projects and their code (like Appium). Since NPM is included by default in NodeJS, what you need is to download and install NodeJS in your system [14]
2. **Install Appium with npm:** Once NPM is installed in your system, you can install the Appium project by opening a command prompt and typing:
   `npm i -g appium@next`
   After the installation, verify you are using some Appium version 2.0.0 by typing:
   `appium --version`
3. **Install Appium uiautomator2 driver:** Appium is already installed with NPM, but you also need to install in Appium a mobile driver which allows you to connect and interact with mobile environments. You can type the command:
   `appium driver install uiautomator2`
   Finally, to verify the driver installation, you can type:
   `appium driver list --installed`

Once everything is installed correctly, we will be able to launch the Appium server in a command prompt by executing:
`appium --relaxed-security --base-path /wd/hub`
For security reasons, Appium client sessions can not request feature enablement via ca-

[12] https://github.com/appium/appium-desktop
[13] https://appium.io/docs/en/2.0/
[14] https://nodejs.org/en/download

pabilities [15]. However, for local testing purposes, we are using the `--relaxed-security` parameter to disable this security temporally. Furthermore, we use the `--base-path /wd/hub` to enable the path direction on which TESTAR requests the connection with Appium.

```
npm
Microsoft Windows [Versión 10.0.19045.2846]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\FernandoTESTAR>appium --relaxed-security --base-path /wd/hub
[Appium] Welcome to Appium v2.0.0-beta.66
[Appium] Non-default server args:
[Appium] {
[Appium]   basePath: '/wd/hub',
[Appium]   relaxedSecurityEnabled: true
[Appium] }
[Appium] Attempting to load driver uiautomator2...
[debug] [Appium] Requiring driver at C:\Users\FernandoTESTAR\.appium\node_modules\appium-uiautomator2-driver
[Appium] Appium REST http interface listener started on 0.0.0.0:4723/wd/hub
[Appium] Available drivers:
[Appium]   - uiautomator2@2.18.0 (automationName 'UiAutomator2')
[Appium] No plugins have been installed. Use the "appium plugin" command to install the one(s) you want to use.
```

## 11.3 Testing a local Android Application Package

The Android Application Package (APK) files are software packages with dependencies that allow the execution of Android applications in mobile environments. These are similar to the `.jar` files we used to run Desktop Java applications such as the Calculator. If we have the desired APK to be tested locally on our host computer, we can configure TESTAR to install and launch the application through Appium in the virtual emulator.

The file `DesiredCapabilities.json` is an additional configuration file for Android protocols we need to use to indicate to TESTAR:

- to which `deviceName` we want to connect (AVD name created).
- the `platformName` that corresponds with the mobile platform (Android).
- the `app` to test (local APK).
- the `automationName` that corresponds with the Appium mobile driver (UiAutomator2).
- the `appWaitActivity` to use a starting state to initialize the testing process.

DesiredCapabilities.json

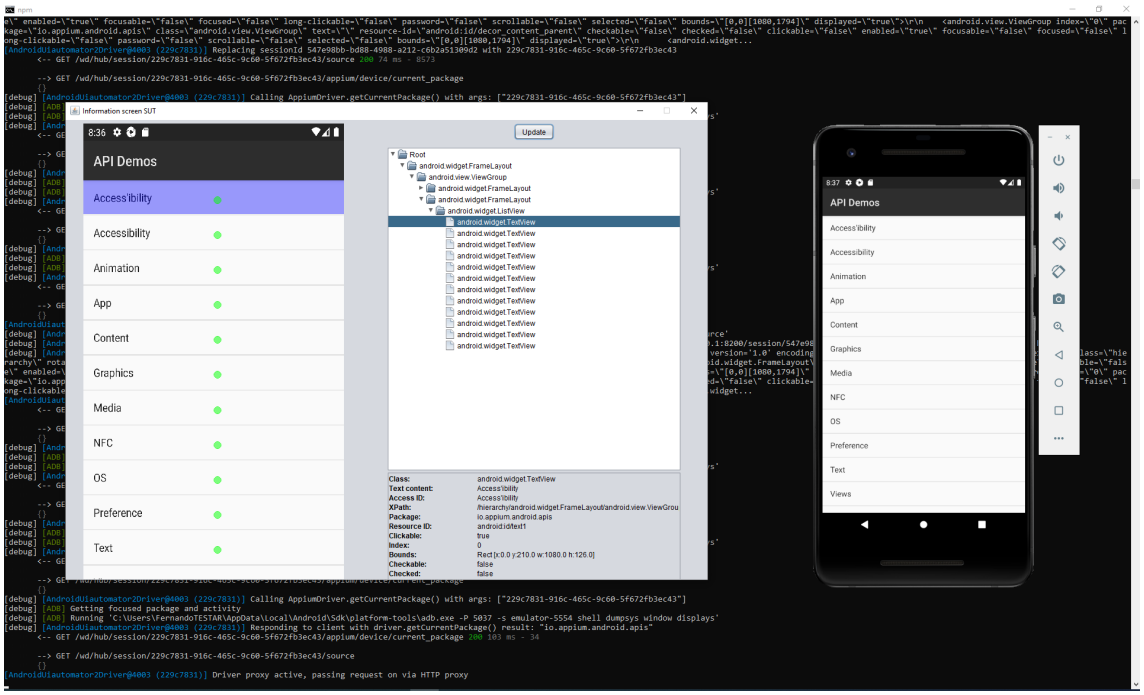
```
{
  "deviceName": "Android Emulator",
  "platformName": "Android",
  "app": "suts/ApiDemos-debug.apk",
  "automationName" : "UiAutomator2",
  "allowTestPackages": true,
  "newCommandTimeout": 500000,
  "appWaitActivity": "io.appium.android.apis.ApiDemos",
  "autoGrantPermissions": false
}
```

[15] https://appium.io/docs/en/2.0/guides/security/

You will find this file together with the Java protocol and test.settings file in the specific protocol directory:

```
testar/bin/settings/android_generic
-- testar/bin/settings/android_generic/Protocol_android_generic.java
-- testar/bin/settings/android_generic/test.settings
-- testar/bin/settings/android_generic/DesiredCapabilities.json
```

Since, for the Android system, the application runs in an external mobile device or a virtual environment, the SPY mode of TESTAR cannot easily draw the information about the detected widgets and derived actions on the computer screen. For this reason, for Android protocols, TESTAR will execute an additional Java dialog that displays an image of the current state in a left panel and the hierarchy of the existing widgets information in a right panel.




**hands-on 48**

SPY a local Android APK with TESTAR through Appium

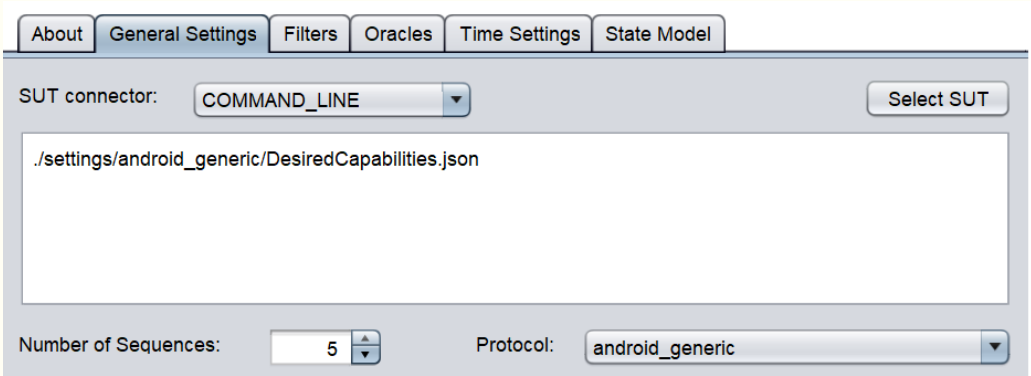
① Verify there is an Android emulator (AVD) running in your system

② Verify there is an Appium server running in your system with the `--relaxed-security` and `--base-path /wd/hub` parameters

③ Open the `testar/bin/settings/android_generic/DesiredCapabilities.json` file. Because TESTAR already provides a demo `ApiDemos-debug.apk` for learning purposes, you only need to take care of modifying the `"deviceName"` JSON value with the AVD name created in your computer.

④ Launch TESTAR and change to the `android_generic` protocol, then, Run the

SPY mode. If you did not change any other setting, the `COMMAND_LINE` connector of TESTAR is already pointing to the file we configured in the previous steps `./settings/android_generic/DesiredCapabilities.json`.




⑤ Play with the left and right panels created by TESTAR. You can manually interact with the application inside the AVD emulator, then TESTAR will automatically update the displayed state.

To generate test sequences, we also need to have the Android device/emulator with Appium as middleware architecture. The `Suspicious Tags` Oracles are configurable in a similar manner to Desktop and Web protocols. And the information generated along the sequence is also created as an HTML report in the output folder `testar/bin/output`. Compared with SPY mode, in GENERATE mode, TESTAR will not execute any additional dialog but just execute actions in the Android application.

**hands-on 49**

Generate test sequences of a local Android APK

① Verify there is an Android emulator (AVD) running in your system

② Verify there is an Appium server running in your system with the `--relaxed-security` and `--base-path /wd/hub` parameters

③ Verify the `testar/bin/settings/android_generic/DesiredCapabilities.json` file is configured

④ Launch TESTAR and change to the `android_generic` protocol. Because we already configured the protocol on previous exercises, we focus on generating some test sequences. Run 5 sequences of 10 actions with the GENERATE mode.

⑤ At the end of the test sequences process, open the HTML reports to check the results.

## 11.4 Testing a remote Android Application Package

It is also possible to indicate TESTAR and Appium to download and test an APK accessible through the internet or hosted inside our company environment. To do this, we only need to configure the `app` values in the `DesiredCapabilities.json` file with the desired URL containing the remote APK.

**hands-on 50**

Configure TESTAR to test a remote APK

① Verify there is an Android emulator (AVD) running in your system

② Verify there is an Appium server running in your system with the `--relaxed-security` and `--base-path /wd/hub` parameters

③ Open the `testar/bin/settings/android_generic/DesiredCapabilities.json` file. Now modify the existing `app` value that points to the local APK `suts/ApiDemos-debug.apk` to the next remote URL that also contains the same ApiDemos application:

`https://github.com/appium/java-client/raw/master/src/test/resources/apps/ApiDemos-deb`

④ Launch TESTAR and change to the `android_generic` protocol. Because we configured the `DesiredCapabilities.json` file, we don't need to modify anything more in the `COMMAND_LINE` connector of TESTAR. Run the SPY mode to verify TESTAR launches the remote APK and the new Java dialog still detects all widgets.

SECTION A

# Troubleshooting with Java versions

**PATH 1**: If `JAVA_HOME` is not defined in your environment variables, you will see the following error message:

```
JAVA_HOME is not set and 'java' command could be found in your PATH
```

**PATH 2**: If your `JAVA_HOME` path does not point correctly to the installation path for the Java Development Kit, you will see the following error message:

```
JAVA HOME is not properly aiming to the Java Development Kit
```

**PATH SOLUTION**: To set your `JAVA_HOME` look for example on stackoverflow[16] for the instructions for your type of machine.

**VERSION 1**: If TESTAR was compiled with a different Java version, you will see the following error message:

```
Exception in thread "main" java.lang.UnsupportedClassVersionError
```

**VERSION SOLUTION**: To check which Java version has been used to compile TESTAR, you can check, for example, this baeldung guide[17].

[16] `https://stackoverflow.com/questions/2619584/how-to-set-java-home-on-windows-7`
[17] `https://www.baeldung.com/java-find-class-version`

SECTION B

# Windows Screen Scaling Settings

In this appendix section, we are going to explain a known TESTAR issue regarding Windows screen display settings and widget coordinates detection.

If with a TESTAR Spy mode run you have been able to correctly detect the widgets with the mouse, by moving it to widgets screen coordinates, you probably won't find this problem in the future, but it is better to keep it in mind to know how to solve it.

We have detected that the Windows screen display settings affect TESTAR when processing the SUT coordinates and their corresponding widgets. We currently know that these two options cause the problem (see Figure 23):

- Display scale % setting
- Advanced scaling setting: Let Windows try to fix apps so they're not blurry.

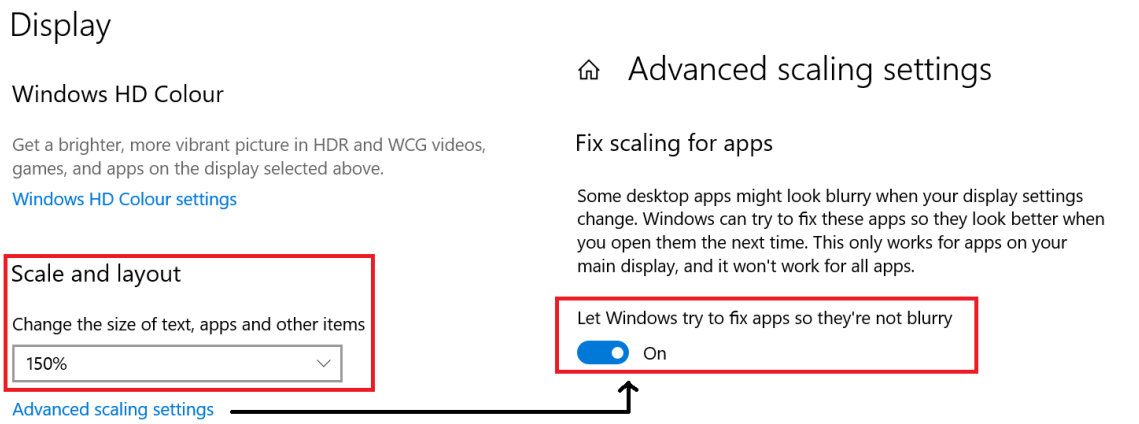



Figure 23: Windows screen scale settings

After checking this issue, we start calling the Windows system to get the current value of the display scale to be applied to the coordinates of the widgets. However, we detected that the coordinates behavior differs in different environments with the same display and blurry options. That's why we have not yet developed a general solution that solves the issue in every Windows environment.

Our current recommendation is to use the 100% display setting value and try disabling the blurry option in case this error persists. `NOTE:` If you are working with a VM, the display settings values from the host computer are transmitted to the VM. So basically you need to change the value in your host computer settings.

Webdriver implementation (see section 8) uses the selenium framework to obtain the SUT information differently than the Windows Accessibility API used for desktop applications. We have again detected different behavior for different environments, but also comparing Windows API vs Webdriver implementations in the same environment with the same scale settings.

That is why we have included a setting configuration in the webdriver protocols, that allows overriding the default scale value in case of continuing with faulty coordinate detection. In the figure 24, we can see the `Override display scale`, on which we can introduce a decimal value as (1.0 - 100%, 0.5 - 50%, 1.5 - 150%) to override the default value taken from Windows system.

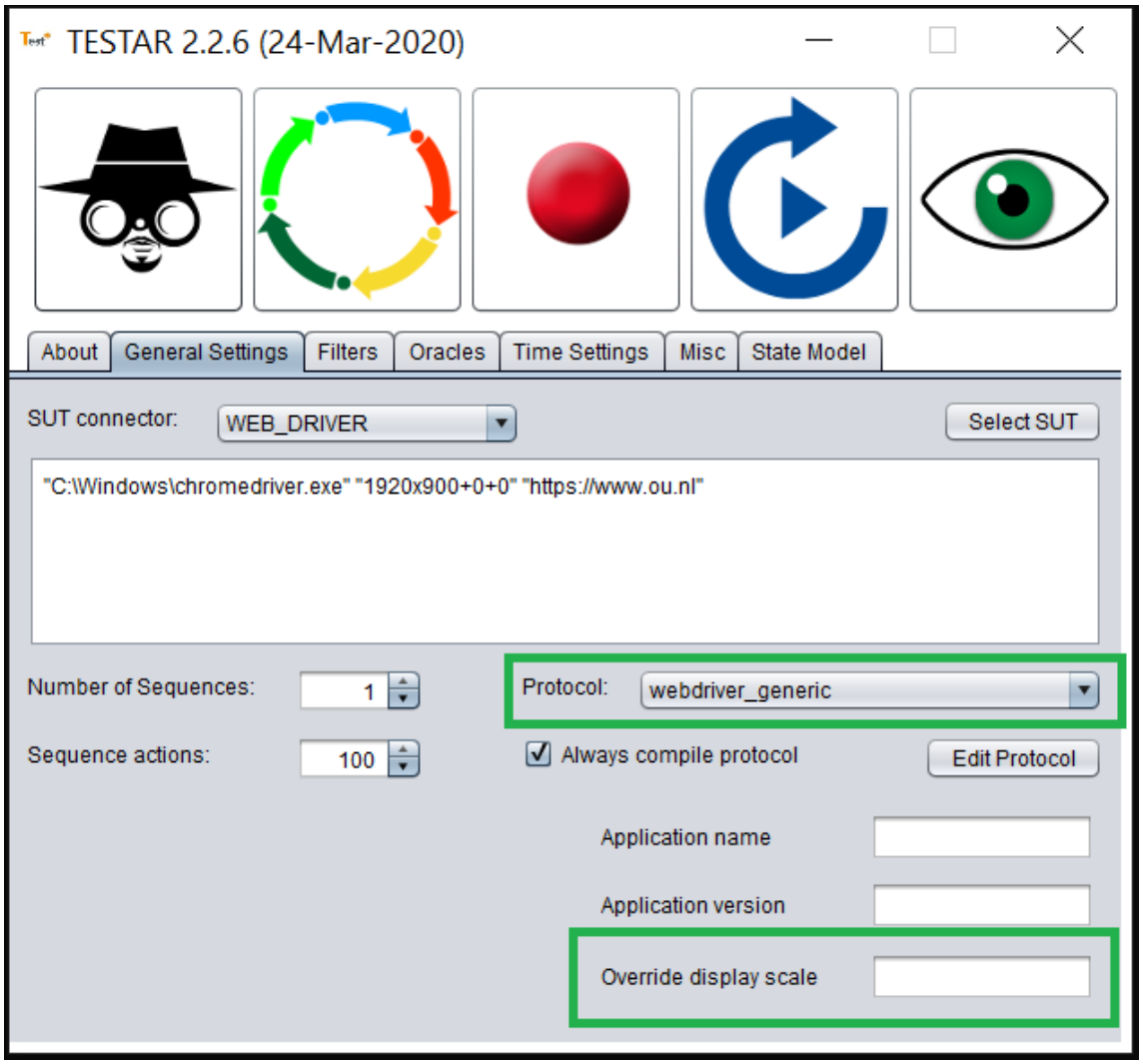



Figure 24: TESTAR Override display scale setting

SECTION C

# ActionDuration test.setting

**ActionDuration** is a non-negative decimal setting that indicates to TESTAR the speed, in seconds, at which an UI action is performed.

When the value of this setting is `0` or `0.0`, TESTAR **teleports** the mouse from the current screen location to the widget coordinates. If this action **teleport** fails, it can be because the Screen scale value is not 100% (see APPENDIX B).

In case the value of this setting is higher than `0.1`, TESTAR performs a mouse movement. However, moving the mouse over a GUI may provoke changes in the GUI widgets and alter the running sequence:

`https://github.com/TESTARtool/TESTAR_dev/issues/224`

SECTION D

# CAPS LOCK event for SPY mode filtering

`Temporally workaround:` Due to this problem sending the CAPS_LOCK key event to a Virtual Machine, we added an optional workaround also to be able to use the ALT key event to enable the SPY filtering mode.

The usage of the button CAPS_LOCK is a useful TESTAR feature to filter undesired action-widgets when using the SPY mode(see Section 6.2).

As you know, we recommend running TESTAR in a Virtual Machine to prevent users to suffer undesired actions in their host computers. Unfortunately, the usage of a Virtual Machine may affect the usage of the CAPS_LOCK event-button.

Normally, when working on a host computer, pressing CAPS_LOCK creates an event in the host system indicating that the button must remain toggled. However, in some Virtual Machine services, this event may not be detected correctly. This means that even if you press the CAPS_LOCK button in your host, the Virtual Machine system will not toggle the button.

SECTION E

# What is a regular expression and what it can do?

A regular expression (regex) is a sequence of characters that defines a search pattern. It can be used to match, search, and manipulate text. Regular expressions are commonly used in programming, text editors, and command-line utilities to perform string manipulation and pattern matching.

Before you start writing a regular expression, you need to identify what pattern you want to match. This could be a specific word, a group of words, a certain character, or a combination of different elements.

The syntax of regular expressions varies slightly depending on the tool or programming language you are using, but the basic principles are the same. Regular expressions are made up of special characters and metacharacters that represent different patterns.

Some common meta-characters include:

. - Matches any single character except a newline character.
* - Matches zero or more occurrences of the previous character or group.
+ - Matches one or more occurrences of the previous character or group.
? - Matches zero or one occurrence of the previous character or group.
— - Alternation operator, matching either the left or right expression.
() - Used to group expressions together.

DESCRIBE rules that we apply in the checkers `https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html`

More information can be found at the following link:

`http://www.vogella.com/tutorials/JavaRegularExpressions/article.html`

`https://zeroturnaround.com/rebellabs/java-regular-expressions-cheat-sheet/`

## E.1 Regex mastery

Regex is a flexible and powerful mechanism that we use in TESTAR to filter undesired action-widgets or to customize Suspicious Tags Oracles. With experience, you can little by little improve your mastery using Regex with TESTAR.

Sometimes, for example, you need to prepare a TESTAR protocol to derive action-widgets in specific buttons of your application. In those cases, maybe you want to apply "inverse" filtering `^(?!(.*tan.*)$).*$` that filters everything except your desired buttons.

For Suspicious Tags, displaying decimal values with more than 2 decimal digits can be considered an error. Regex expression as `\d+\.\d{3,}` can help you to detect these cases.

SECTION F

# Keyboard actions and the CompoundAction builder

We can also define specific input actions by specifying a sequence of Keyboard Actions to navigate and input data in the required fields.

To enable keyboard navigation we will use a `CompoundAction` builder, that contains a series of actions that you want to execute when the system starts up. Note that, in the following example, when the username is being typed, it is assumed that keyboard focus is on the user field, but this may not be the case when you start up. So you might first need to TAB your way to the right widget.

```
new CompoundAction.Builder()
    // assume keyboard focus is on the user field
    .add(new Type("john"),0.1)

    // assume next focusable field is pass
    .add(new KeyDown(KBKeys.VK_TAB),0.5)
    .add(new Type("demo"),0.1)

    // assume login is performed by ENTER
    .add(new KeyDown(KBKeys.VK_ENTER),0.5).build()

    .run(sut,null,0.1);
return sut;
```

Sometimes we want to enter special characters into a text field, the writing of special characters into text fields depends on the user Keyboard language. The following is an example of how to write the `@` character from an ENG Keyboard.

```
import org.testar.alayer.devices.AWTKeyboard;

Keyboard kb = AWTKeyboard.build();

//Based on ENG Keyboard, Shift + 2 typing @ character
kb.press(KBKeys.VK_SHIFT);
kb.press(KBKeys.VK_2);
kb.release(KBKeys.VK_2);
kb.release(KBKeys.VK_SHIFT);
```

SECTION G

## Failure BINGO!

After doing the hand-on in Section 6 you know how to use the tool, your task is to setup a longer test for the Calculator, e.g. 30 sequences with a length of 50 actions (feel free to use different values). Run the tool and observe its output! Does it find and report the failures? Can you replay the sequences that found failures and can you reproduce the failures? If the tool executes undesirable actions, improve your setup and restart the test. At the end of this task you should have a folder with several erroneous sequences that can be replayed and expose failures.

Find as many failures as possible and fill the following Bingo card.


Yell **BINGO!!** if you are the first to fill the card. The BINGO is valid if the "How to reproduce fields" can really reproduce the bug. Note that this can be done manually or with TESTAR replay.

SECTION H

# Keyboard shortcuts

Several keyboard shortcuts are available for the different TESTAR working modes.

| Keyboard shortcut | Effect | Working modes | | |
|---|---|---|---|---|
| | | SPY | GENERATE | REPLAY |
| Shift + Arrow Up | Show widgets info | ✓ | | |
| Shift + Arrow Down | Close the mode and go back to TESTAR dialog | ✓ | ✓ | ✓ |
| Shift + Space | Toggle slow motion test | | ✓ | ✓ |
| CAPS_LOCK/TAB + (Shift) Ctrl | UI widgets' actions filtering | ✓ | | |

Table 1: Keyboard shortcuts

Figure 25 gives an overview of how to go from mode to mode and back to TESTAR. A summary of the shortcuts in Appendix H. It is a good idea to become familiar with this Mode and its shortcuts.

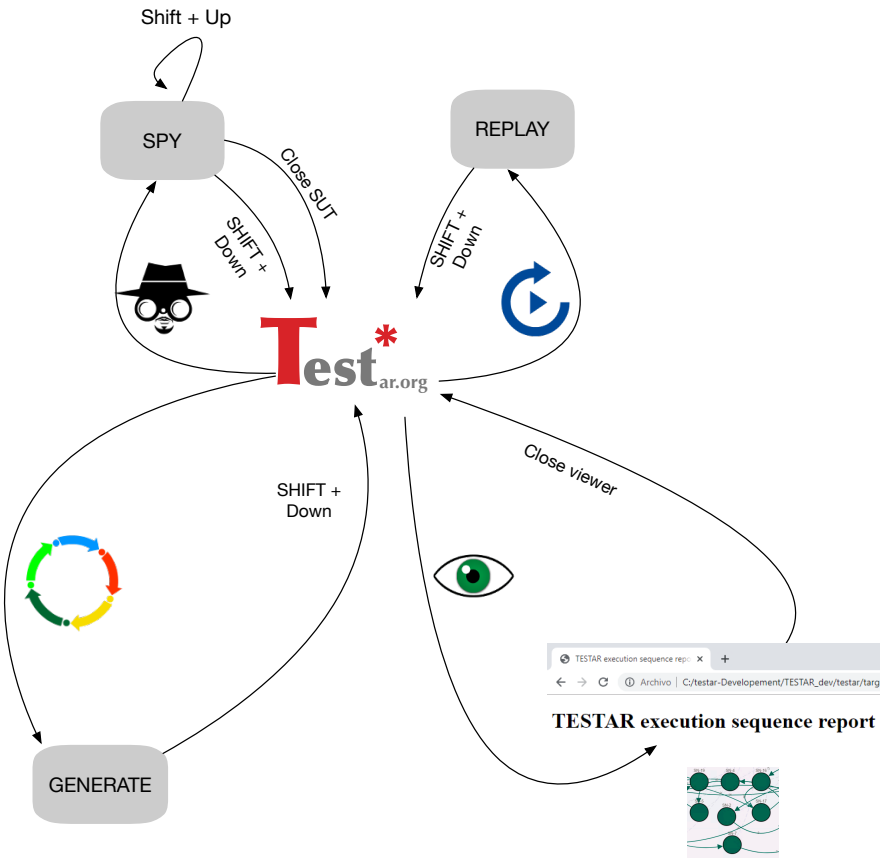



Figure 25: TESTAR Modes

**SECTION I**

# Directories

| | |
|---|---|
| ./settings | Tests set ups |
| ./output | Reports of the different TESTAR runs |
| ./output/`timestamp_SUTname` | Reports: logs, screenshots, graphs, metrics, serialized tests |
| ./../sequences | All the serialized test sequences |
| ./../sequences_V | Classified test sequences by verdict V. Where V can be from {ok, suspiciostitle, unexpectedclose} |
| ./../HTML reports | Visual html report |
| ./../srcshots | Screenshots of tests UI states and executed UI actions |
| ./../logs | Tests logging data |
| ./output/graphs | Tests graphing for visual analysis |
| ./output/metrics | Tests performance indicators |
| ./output/temp | Temporary files such as the last recorded test sequence |

SECTION J

# Test settings

- **ActionDuration** = a non-negative decimal. Sets the speed, in seconds, at which an UI action is performed. For example, typing a text will introduce delays between each key stroke.
- **AlgorithmFormsFilling** = true or false. Enables or disables a specific UI action selection algorithm that will try to populate data in UI forms.
- **ClickFilter** = regular expression. Prevents UI actions to be performed on UI elements whose *TITLE* matches the regular expression. The rationale behind this is that certain UI actions might be dangerous or undesirable without human supervision (e.g. printing documents, files operations). For example: .*[cC]lose.*|.*[eE]xit.*|.*[pP]rint.*).
- **CopyFromTo** = (source_file_path;target_file_path)*. A list of (>=0) pairs of source and target files to copy before a test starts (click the text-area and a file dialog will pop up). Sometimes, it can be useful to restore certain configuration files to their default prior to SUT execution, so that the SUT starts from a desired state.
- **Delete** = (file_path)*. A list of (>=0) files to delete before a test starts (click the text-area and a file dialog will pop up). Certain SUTs may generate configuration files, temporary files and/or files that save the SUT's state. Thus, you can restore your SUT environment to a desired state removing files generated from previous executions.
- **DrawWidgetInfo** = true or false. Sets whether to display detailed overlay information, inside the SPY mode over the selected widget in UI of the SUT.
- **DrawWidgetTree** = true or false. Sets whether to display a graphical representation of the widget-tree, inside the SPY mode for the selected widget in the SUT's UI.
- **DrawWidgetUnderCursor** = true or false. Sets whether to display brief overlay information, inside the SPY mode over the selected widget in the UI of the SUT.
- **ExplorationSampleInterval** = a positive number. Sets the metrics sampling interval by the number of executed UI actions during a test.
- **ForceForeground** = true or false. Sets whether to keep the SUT's UI window active in the screen (e.g. when its minimised or when a process is started and its UI is in front, etc.).
- **ForceToSequenceLength** = true or false. Setting the value to true, if a test fails (e.g. the SUT crashes), TESTAR continues the test sequence until it reaches the specified test sequence length (check *SequenceLength* property). Otherwise (false value), the test will finish in the presence of a fail.
- **MaxTime** = a positive number. Sets a time window, in seconds, after which the test is finished (e.g. stop after an hour, a day or a week).
- **Mode** = SPY, GENERATE, VIEW or REPLAY (check *ShowVisualSettingsDialogOnStartup* property). Runs the tool into the SPY, GENERATE, VIEW or REPLAY mode.
- **NonReactingUIThreshold** = a positive number. Sets a test window (number of UI actions) for which a non-reacting UI will force to perform UI actions that could potentially make the UI to react (e.g. an ESC key stroke to close a popup dialog box).
- **LogLevel** = 0, 1 or 2. Sets the logging level to critical messages (0), information messages (1) or debug messages (2).

- **OnlySaveFaultySequences** = true or false. Sets whether to save non-fail test sequences.
- **ProcessesToKillDuringTest** = regular expression. Any process name that matches the regular expression and is started during a test will be automatically killed. The rationale behind this is that some UI actions could start undesirable processes (e.g. an email client). For example: (e.g. .*[oO]utlook.*|firefox.exe).
- **ProtocolClass** = settings_folder/Test_protocol_class_name. Links to the test protocol class under the folder denoted by the *MyClassPath* property.
- **ReplayRetryTime** = a positive number. Inside the replay mode, establishes the time window in seconds for trying to replay a UI action of a replayed test sequence.
- **SequenceLength** = a positive number. Sets each test sequence (check *Sequences* property) length as the number of UI actions to perform[18]. Check the *StopGenerationOnFault*, *ForceToSequenceLength*, *MaxTime*, *SuspiciousTitles*, *TimeToFreeze* and *ProtocolClass* properties for specific behaviour.
- **Sequences** = a positive number. Number of times to repeat a test.
- **ShowVisualSettingsDialogOnStartup** = true or false. Sets whether to display the tool UI. If false is used, then the tool will run in the mode of the *Mode* property.
- **StartupTime** = a positive number. Sets how many seconds to wait for the SUT to be ready for testing (its UI being accessible by TESTAR). If the SUT did not start on time the test will not run. Otherwise, test will start as soon as the UI is accessible. Take into account that the first time the SUT is run on your environment will usually take more time than next executions (e.g. due to memory catching).
- **StopGenerationOnFault** = true or false. Sets whether to finish a test in the presence of a fail (e.g. a SUT crash). Setting it to false does not necessarily mean that the test will continue, but the test will try to continue as far as the SUT accepts additional UI actions and the test set up does not finish the test by other means (e.g. *MaxTime*, *SuspiciousTitles*, *TimeToFreeze* or *ProtocolClass* properties).
- **SUTConnector** = COMMAND_LINE, SUT_WINDOW_TITLE or SUT_PROCESS_NAME. Sets the approach used to connect with your SUT:
  - COMMAND_LINE: *SUTConnectorValue* property must be a command line that starts the SUT. It should work from a Command Prompt terminal window (e.g. *java -jar SUTs/calc.jar*). For web applications follow the next format: *web_browser_path SUT_URL*.
  - SUT_WINDOW_TITLE: *SUTConnectorValue* property must be the non-empty *title* displayed in the SUT's main window. The SUT must be manually started and closed.
  - SUT_PROCESS_NAME: *SUTConnectorValue* property must be the process name of the SUT. The SUT must be manually started and closed.
- **SUTConnectorValue** = check *SUTConnector* property.
- **SuspiciousTitles** = a regular expression. Checks the UI for any suspicious title that could denote problems in the SUT. TESTAR checks whether there exists a widget' *TITLE* in the UI that matches the regular expression. If a match was found the test will continue but you will find the issues found in the reports. For example, a critical message like "A NullPointerException Exception has been thrown" can be represented by the regular expression ".*NullPointerException.*".
- **TimeToFreeze** = a positive number. Sets the time window, in seconds, for which to wait for a not responding SUT. After that, the test will finish with a fail. The rationale behind this is that the SUT could hang, be performing heavy computations or be waiting for slow operations (e.g. bad internet connection). The value of the

[18]Note: higher values will consume more hardware resources, specialy if graphing was activated.

property is thus a threshold after which the SUT is interpreted to have hung.

- **TimeToWaitAfterAction** = a non-negative decimal. Sets the delay, in seconds, between UI actions during a test. It directly affects the reproducibility of tests and tests performance. Setting it to a low value will speed up the tests, but the SUT could not have finished processing an action before the next action is executed by TESTAR. In the latter case the test could not be reproducible, but it could reveal potential faults (stress testing).
- **UseRecordedActionDurationAndWaitTimeDuringReplay** = true or false. Inside the replay mode sets whether to use the action duration (check *ActionDuration* property) and action delay (check *TimeToWaitAfterAction* property) as specified in the recorded test sequence. If set to false, the values from the current set up are used.
- **VisualizeActions** = true or false. Sets whether to display overlay information, inside the SPY mode, for all the UI actions derived from the test set up.

# Index