# Ripping Process

The following four steps have to be executed by clicking the corresponding buttons.

**Device Boot**In the Device Boot step, the device emulator is preliminary booted. When operative system booting is completed, the user can proceed to the next step

**Deploy**In the Deploy step, the code of the Android application under test is instrumented by the Emma library (for code coverage assessment) and deployed into the AVD emulator tool together with the GUI Ripper itself. Successively, an image of the Android system, called *Snapshot* image, is created, in order to be restored after each ripping step execution. It is recommended that the AVD has just be created for this execution.

**Define Snapshot**By clicking on this button, the emulator is re-opened and the user has to fix the initial state of the device and of the application that will be the starting point for any future test case. When the desired starting interface is reached, the user has to close the emulator: a snapshot of the device image will be autoimatically closed.

**Run Ripper**By clicking this button the Ripping process is started. This step of the process is completely automatic and it can last many hours for complex applications. No user interactions are needed until the execution is terminated.

**Generate Reports**By clicking this button, the collected data are elaborated and outputs such as the Execution Report, the generated Test cases, the GUI Tree and the Code Coverage report are generated.

**Open Results Folder**It opens the folder with all the results files.

The **Restore** option can be used if we want to continue an incomplete ripping. In this case, we have to check Restore and to click the Run Ripper button

# Input files description

Many parameters influencing the execution of the ripper can be set.

The file **preferences\_Ripper.xml** contains parameters influencing the execution of the Ripper when it performs a systematic exploration of the GUIs.

The file **preferences\_Random.xml** contains parameters influencing the random exploration.

Both the preferences files have the same structure.

## Structure of xml preferences files

* [General parameters](http://wpage.unina.it/ptramont/GUIRipperConfigurable_Java7.htm#General_parameters)
* [Automation parameters](http://wpage.unina.it/ptramont/GUIRipperConfigurable_Java7.htm#Automation_parameters)
* [Scheduler parameters](http://wpage.unina.it/ptramont/GUIRipperConfigurable_Java7.htm#Scheduler_parameters)
* [Interactions parameters](http://wpage.unina.it/ptramont/GUIRipperConfigurable_Java7.htm#Interactions_parameters")
* [Planner parameters](http://wpage.unina.it/ptramont/GUIRipperConfigurable_Java7.htm#Planner_parameters)
* [Comparator parameters](http://wpage.unina.it/ptramont/GUIRipperConfigurable_Java7.htm#Comparator_parameters)
* [Strategy parameters](http://wpage.unina.it/ptramont/GUIRipperConfigurable_Java7.htm#Strategy_parameters)

## General parameters

**PACKAGE\_NAME**: String

* The name of the package that contains the classes (specifically, the activities) which make up the application under test.
* Technically, this parameter defines the target package for the Instrumentation; as a result, any attempt to exercise items of software residing outside this package will result in a Runtime Exception and a "dead" branch of the Gui Tree.

**CLASS\_NAME**: String

* The fully qualified name (i.e. including the package) of the class of the start activity of the application under test.
* Every trace in the ripping session will have this activity as its starting point.

**RANDOM\_SEED**: long {0 = random}

* This value is used to setup the random number generation mechanism. Several instances of this are used with various purposes such as: filling in the values of text fields, randomizing the sequence of actions to be performed when using the Random engine, and so on...
* When initialized with the same seed, the Ripper should be able to generate the same sequence of random numbers, allowing the exact repetition of a session.
* When set to zero, a random seed will be generated at startup based on the system time.

## Automation parameters

Following are the entries of the map stored in the automation sub-node (direct child of the main node) of the XML file.

**SLEEP\_AFTER\_EVENT**: int

* How many milliseconds to wait after performing each of the actions of which a task is comprised of.

**SLEEP\_AFTER\_RESTART**: int

* How many milliseconds to wait at restart, that is before performing any of the actions in a task.

**SLEEP\_AFTER\_TASK**: int

* How many (additional) milliseconds to wait at the end of a task, after having performed all the actions it is comprised of.

**SLEEP\_ON\_THROBBER**: int

* How many milliseconds to wait after an event if a throbber (spinning wheel displayed when the user is supposed to wait) is on the screen.
* Note that the Ripper will always wait **SLEEP\_AFTER\_EVENT** ms even if the throbber disappears before this amount of time. Then, it waits for the throbber to disappear; if **SLEEP\_ON\_THROBBER** ms pass, the Ripper proceeds to the next event, even if the throbber is still displayed.

**SCREENSHOT\_FOR\_STATES**: boolean

* When true, an image capture of the display screen is performed after the last action in a task has been performed and saved as a JPEG file.

**SCREENSHOT\_ONLY\_NEW\_STATES**: boolean

* When true, a screenshot is saved at the end of a task only when a new state is found; in other words, when the final state of a task is equivalent to one of the states already discovered (which we already have a screenshot of), no further pictures are taken.
* Ignored if **SCREENSHOT\_FOR\_STATES** = false

**SCREENSHOT\_FOR\_EVENTS**: <not yet implemented>

## Scheduler parameters

Following are the entries of the map stored in the scheduler sub-node of the XML file.

**SCHEDULER\_ALGORITHM**: {BREADTH\_FIRST|DEPTH\_FIRST}

* The strategy used to explore the application under test:
  + with the DEPTH\_FIRST algorithm, the branch starting from a node of the GUI Tree is explored thoroughly before exploring the neighbours of (nodes at the same depth than) that node;
  + with the BREADTH\_FIRST algorithm, on the opposite, all neighbouring nodes are explored before descending to deeper nodes.
* Unused by the Random Ripper.

**MAX\_TASKS\_IN\_SCHEDULER**: int

* Tasks to be performed during the ripping session are planned in advance and stored in a task list. This parameter limits the dimension of this list.
* This is useful when using the Random ripper, since it can generate very long sequence of actions, leading to long I/O operations and possibly "Out of Memory" errors.

## Interactions parameters

*Following are the entries of the map stored in the interactions sub-node of the XML file.*

**EVENTS**: String []

* Defines the type of the interactions to be fired by the Ripper as events, and the types of the widget upon which to fire them.
* Each String in the array will start with the Interaction Type of the event to fire, followed by the Simple Types of the widgets on which the event should be fired, separated by a comma. Spaces will be ignored.
* Example: "click, button, image, menuItem"

**INPUTS**: String []

* Defines the type of the interactions to be fired by the Ripper as inputs and the types of the widget upon which to fire them.
* The format is the same as for the **EVENTS**.

## Planner parameters

Following are the entries of the map stored in the planner sub-node of the XML file.

**MAX\_EVENTS\_PER\_WIDGET**: int {0 = infinite}

* Limits the number of times an event is generated when iterating on a group widget.
* For instance, when **MAX\_EVENTS\_PER\_WIDGET** = 0 the selectListItem interaction generates a select event (scroll, focus and click) for each entry in a List View.
* If **MAX\_EVENTS\_PER\_WIDGET** = 5, only the first five elements in the List View (including separators) will be selected.

**PLANNER**: String {SimplePlanner}

* Currently, the unique possible value is "SimplePlanner"

**TEXT\_VALUES\_ID\_HASH**: boolean

* To be set to true

## Comparator parameters

*Following are the entries of the map stored in the comparator sub-node of the XML file.*

**COMPARATOR\_TYPE**: String

* This parameter defines the state equivalence criterion. Allowed values are:
  + NullComparator: no comparation, two states are always different (this is the default)
  + NameComparator: two states are equivalent if they share the same name (they are instances of the same Activity)
  + CustomWidgetsComparator: two states are equivalent if they share the same set of widgets (each widget in state A is equivalent to a widget in state B and viceversa)
  + CustomWidgetsDeepComparator: two states are equivalent if they share the same set of widgets, and the equivalent widgets share the same properties
* For instance, if state A has an Edit Text with id 1234, CustomWidgetsComparator requires that state B has an Edit Text with id 1234 in order to consider B equivalent to A. CustomWidgetsDeepComparator additionally requires that the Edit Text has the same hint text.

**COMPARE\_STATE\_TITLE**: boolean

* When true, NameComparator also checks whether two states have the same title. That is: two states are the same if they share the same name and title.
* If **COMPARE\_ACTIVITY\_NAME** = true, this also applies for the other Comparator types.

**COMPARE\_LIST\_COUNT**: boolean

* When true CustomWidgetsComparator and CustomWidgetsDeepComparator will consider two List View to be equivalent only if they have the same size.

**COMPARE\_MENU\_COUNT**: boolean

* When true CustomMenuComparator and CustomWidgetsMenuComparator will consider two Menu to be equivalent only if they have the same size.

**COMPARE\_VALUES**: boolean

* When true, CustomWidgetsDeepComparator will consider two widgets to be equivalent only if they have the same value.
* For instance, if state A has a Rating Bar with id 4567, name "Vote for this book" and sat to the value of "4 stars", state B will not be equivalent to A unless it has a Rating Bar with the same id and the same name which is set to the same value.

## Strategy parameters

*Following are the entries of the map stored in the strategy sub-node of the XML file.*

**MAX\_NUM\_TRACES**: int {0 = infinite}

* Terminates the ripping session after processing this amount of tasks. 0 value is not admitted for Random Ripping.

**TRACE\_MAX\_DEPTH**: int {0 = infinite}

* Limits the exploration of the GUI to a certain depth of the resulting GUI Tree.
* When **TRACE\_MAX\_DEPTH** = n>0, GUIs resulting from the execution of n actions (that is, nodes at depth n in the GUI Tree) won't be further explored, even when they're new and events could be fired on them.

**EXPLORE\_ONLY\_NEW\_STATES**: boolean

* When true, states that are equivalent to previous one (according to the defined Comparator) won't be explored.
* The Random engine forces **EXPLORE\_ONLY\_NEW\_STATES** = false for obvious reasons.
* When set to false, either **TRACE\_MAX\_DEPTH** or **MAX\_NUM\_TRACES** should be provided. Otherwise, there won't be any termination criterion, and the exploration will continue indefinitely.

## Description of the output files

 At the end of the Run Ripper step, seven files are in the **results** folder:

* **building.txt** containing the echo building activities of the deploy step.
* **test.txt** containing the output of the Ripping step, including the list of executed traces, the testing time, the test results (OK if no crashes or exceptions happened, elsewhere the stack trace of the encountered problem) and a reference to the file containing the coverage data.
* **log.txt**, that is the Android log file of the Ripper execution
* **logf.txt**, that is a filtered version of log.txt containing only messages related to the Ripper
* **build.txt**is used for monitoring the correct building of the application.
* **ripper.txt**, that is used for monitoring for the ripping completion
* **artifacts.txt**, that is used for monitoring for the results generation completion

 In the **output** subfolder there are 7 files:

* **report\_slim.txt**, that contains a summary reporting the number of executed traces, the testing time, the number of happened failures and crashes.
* **extract.txt**, that contains only the description of the stack traces occurred in case of crashes or exceptions.
* **TestSuite.java**, that contains executable JUnit test cases that are able to replicate the exploration done by the GUI Ripper. Please note that the generated test cases are subdivided in traces (i.e. sequence of events from the start to the end of the app) and that test cases must be executed one by one.
* **guitree.dot**, that shows the obtained GUI tree in dot format (viewable with [Graphviz)](http://www.graphviz.org/Download_windows.php" \t "_blank)
* **guitree\_efg.xml**, that contains the Event Flow Graph (as defined by [Atif M. Memon](http://www.cs.umd.edu/~atif/publications.shtml)) obtained on the basis of the information contained in the GUI Tree.
* **guitree\_efg.dot**, that shows the Event Flow Graph in dot format.
* **report.txt**, that contains some aggregative information about the executed ripping.

The **output\coverage** subfolder is directly generated by Emma. By opening the *index.html* file it is possible to explore the obtained coverage in terms of covered classes, methods, blocks and LOCs.

In the **files** subfolder there are 3 files:

* **closed.txt**, that is used for monitoring the completion of the Ripping.
* **guitree.xml**, that contains the GUI tree describing the exploration done by the Ripper.
* **activities.xml**, that contains the description of all the Activity instances occurred during the exploration.

In the **screenshots** subfolder are stored device screenshots corresponding to each of the Activity instances listed in the *activities.xml* file.

The **restore** subfolder is used to maintain temporary information during the ripping. After the completion of the ripping it is deleted.

The **coverage**subfolder contains the coverage files generated by emma: the coverage.em file contains the characteristics of the source code of the application under test, while the coverage.ec files contains information about the coverage reached by any of the explored traces.

For space reasons, the screenshots are held only in systematic ripping executions, not in random ripping.