KLARÄLVDALENS DATAKONSULT AB

Programmer's Manual

KD EXECUTOR

Table of Contents

I. KD Executor Basics .....
1. Overview .....
The Benefits Of KD Executor 3
Operation Modes 4
Reading Instructions 4
2. KD Executor Shell .....
Launching KD Executor 6
Main Window 6
Test Suite Dialog 15
Test Dialog 17
Property Editor 19
Options Dialog 21
Transfer Test Setup 21
3. The KD Executor Sidebar .....
4. Printing Interface Properties .....
5. Using the KD Executor Core Engine .....
Preloading KD Executor 28
KD Executor—Simple Source Code Modification 29
KD Executor—Full Control 31
II. Enhancing KD Executor .....
6. Getting the Best Results with KD Executor .....
Surviving Source Code Changes 36
7. Adapters .....
Developing Your Own Adapters 39
An Example Adapter 40
Putting the Adapter to Work 43
8. Developing Additional Properties .....
An example property extension 44
Putting the Property Extension to work 48
9. Frequently Asked Questions .....
10. The Format Of The Script Files .....
Low-level Events 51
High-level events 54
KD Executor Events 54

List of Figures

2.1. New Test Setup 6
2.2. KD Executor Main Window 7
2.3. The Test Suite Table 9
2.4. The Test Run Table 11
2.5. Detailed Results Tab 12
2.6. Test Suite Dialog 15
2.7. Test Dialog 18
2.8. Property Editor (Test Case) 19
2.9. Property Editor (Test Result) 20
2.10. Options Dialog 21
2.11. Platform Adjustment 22
2.12. Global Rule Definition 22
3.1. KD Executor side bar 24
3.2. Configuring Keybindings 24
3.3. Example Of Specifying Key Bindings For Special Recording Features 26
5.1. KD Executor Configuration Dialog 29
5.2. An Example of an Application Enhanced for Recording 31
5.3. An Example of an Application Enhanced for Playback 32
6.1. Code example for object naming 35
7.1. Adapter Configuration 38
7.2. Header file for sample adapter 40
7.3. Source File For Sample Adapter 41
8.1. Header file for sample property extension 45
8.2. Implementation file for sample property extension 45

Part I. KD Executor Basics

Table of Contents

1. Overview .....
The Benefits Of KD Executor 3
Operation Modes 4
Reading Instructions 4
2. KD Executor Shell .....
Launching KD Executor 6
Main Window 6
Test Suite Dialog 15
Test Dialog 17
Property Editor 19
Options Dialog 21
Transfer Test Setup 21
3. The KD Executor Sidebar .....
4. Printing Interface Properties .....
5. Using the KD Executor Core Engine .....
Preloading KD Executor 28
KD Executor—Simple Source Code Modification 29
KD Executor—Full Control 31

Chapter 1. Overview

The Benefits Of KD Executor

KD Executor is an automation tool which will let you record a script for playing back at a later time. Here are a few examples of when this can be useful:

Automatic Testing	Using KD Executor, you can record a test case which will be executed on a daily basis. This is often referred to as regression testing. Besides just playing back the script, you can also use KD Executor to ensure certain properties of your interface is correct, like a push button being disabled, a list box containing certain items, etc.
Demo versions of your program	Since KD Executor can be compiled into your executable, you can ship a version of your software to a potential customer, that only allows him to see your software run a pre-recorded script.
Trade show demos	Similar to the above example, you can record a demo for an unattended computer at a trade show.
Remote control	To some extent, you can also use KD Executor to control your application remotely. Imagine that you have a customer on the phone asking *How do I* .... Using KD Executor you can ask him to start his program with a special flag, and you can now control it e.g. over the Internet.

In an ideal world, all KD Executor would do is to record raw X11 or Win32 events, and play these back at the users request. This is, however, not sufficient in the real world. The playback scenario might look a bit different than the recording scenario and the above will fail. Examples of this include different font sizes, different layouts, different languages (resulting in different text length), or even the application being played back, having changed slightly since the script was recorded. All these factors result in the user interface components being located at different positions during playback compared to recording. This problem is of special importance when automation is used for regression testing. During regression testing, the layout of the user interface might change in minor or major ways during the lifetime of the recorded script.

KD Executor addresses the problem described above in two ways:

- First, KD Executor records events for each widget individually, rather than for the application as a whole. This means that if you press a button located on position (100,20) during recording, then the button will be pressed during playback, even if it for some reason is now located at position (400,80).

- The second method for addressing the problem with different font sizes, different layout, and different language, is to record the intention rather than the actual action for certain actions on a number of Qt widgets. (These event modifications are referred to as *adapters* in KD Executor.)

  As an example, look at the tab widgets shown below. The first screen shot shows the widget during recording, while the second shows it during playback. During playback the font size is larger than during recording.

  The bar with all the buttons is one single widget. That means that even if the mouse event gets to the correct widget, the position will be wrong within the widget, and as a consequence, the wrong tab will be selected. This is indicated with a cross in both widgets. The cross is at the same coordinates.

  The solution in this situation is to record which tab was selected during recording, and when sending mouse events to the tab widget, ensure that they hit the correct tab.

  See Chapter 7, *Adapters* for a detailed description on how to customize adapters.

Operation Modes

You can use KD Executor in two different ways. For most users who use KD Executor for testing, the *KD Executor Shell* (which is the binary kdxshell) is the most convenient way to use KD Executor. It lets you create, manage, and execute test cases in a friendly graphical test environment. Using the KD Executor Shell is described in the next chapter.

More advanced users who already have a test environment in place can also use KD Executor as an advanced record and playback engine only, using and configuring KD Executor from the command line (This is the binary kdexecutor). This will be described in later chapters.

Reading Instructions

Not everybody needs to read every thing in this manual. The following list will tell you what each chapter contains and who it is meant for.

- Chapter 2, *KD Executor Shell* introduces you to the KD Executor shell, if you do not

plan to use KD Executor for regression testing, you may skip this chapter.

- Chapter 3, *The KD Executor Sidebar* tells you about the KD Executor side bar, which is usefull during recording sessions.

- Chapter 4, *Printing Interface Properties* tells you about the KD Executor property picker which is used during regression testing to ensure properties of you dialogs remain the same over the lifespan of the application under test. If you do not plan to use KD Executor for regression testing you may skip this chapter.

- Chapter 5, *Using the KD Executor Core Engine* tells you how to use the kdexecutor binary as a standalone application (without the shell), and tell you how you can compile KD Executor into your application. In case you are only interested in KD Executor for regression testing, you may skip this chapter.

- Chapter 6, *Getting the Best Results with KD Executor* should be read by those people developing the Qt application on which KD Executor operates. It includes tips on how to write Qt applications which makes KD Executor playback as robust as possible.

- If custom widgets are developed for your Qt application, then to ensure optimal playback, you may wish to develop your own adapters for recording intention rather than action for your widget. This is what Chapter 7, *Adapters* is about. It should be read by the application developers.

- Chapter 8, *Developing Additional Properties* tells you how to develop your own properties for the property picker. This is useful only if you use KD Executor for regression testing. The chapter should be read by the application developer.

- Chapter 9, *Frequently Asked Questions* is a list of frequently asked questions. Everybody should familiarize themselves with this list.

- Finally Chapter 10, *The Format Of The Script Files* tells you about the format of the script file. This is merely a reference for those who wants to generate scripts themselves or who want to parse the scripts for one reason or the other.

Chapter 2. KD Executor Shell

Launching KD Executor

Launching KD Executor can basically be done in two ways:

- With a graphical user interface, to enable you to modify and create test suites, test cases, etc.
- Without a graphical user interface, running directly in the console. Using this, you can let KD Executor execute test cases automatically, without any interference from the user, for example from a cron job or similar.

To launch KD Executor with a graphical user interface, simply run the file `kdxshell`, without any additional parameters. This will start KD Executor with the settings specified within the program, possibly the default.

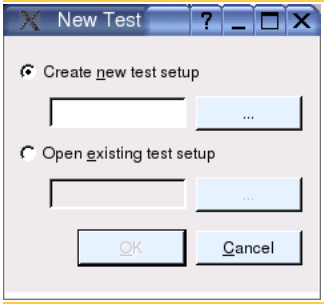
If you want to use the ability to let KD Executor execute test cases automatically, there are a few parameters that needs to be used:

`-setup <setup>`	Will load the specified test setup
`-run all|selected`	Using this parameter, you can specify which test cases you want KD Executor to execute - all, or only the selected.
`-exporthtml <filename>`	This will export the test results of the latest test run to the specified HTML file. It requires the -run parameter.

Main Window

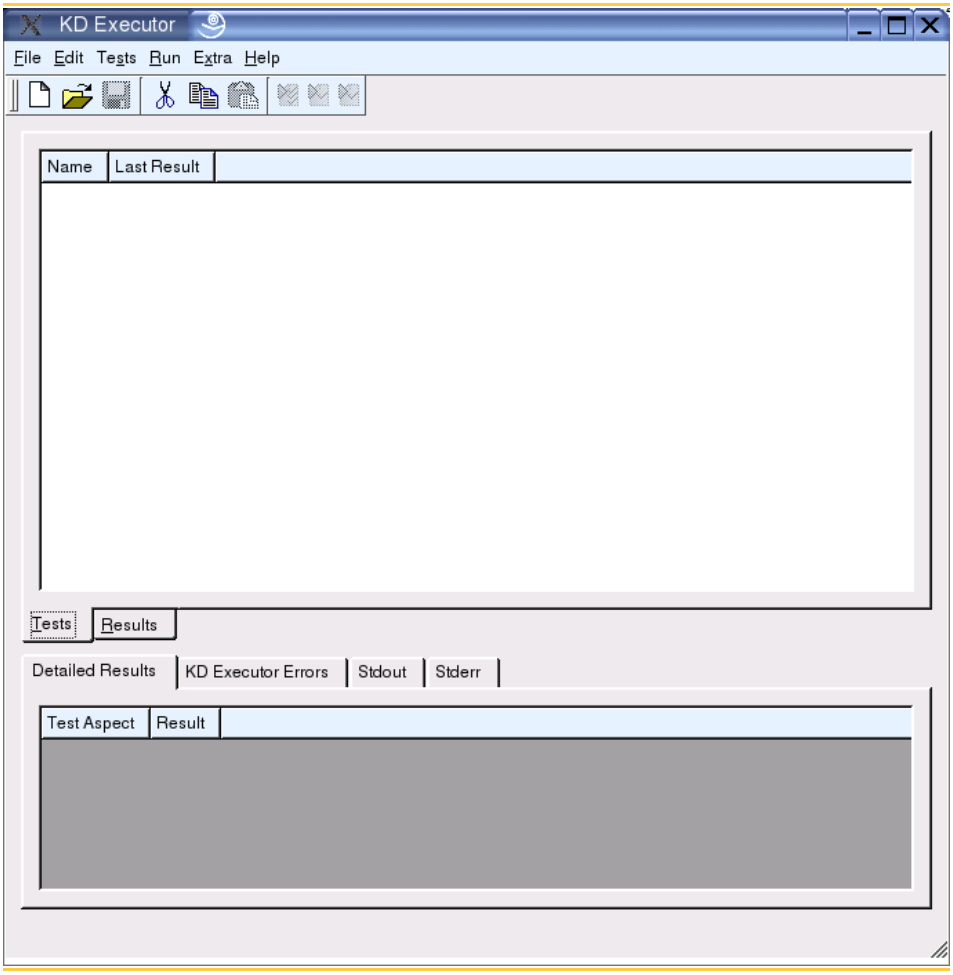
When first launching KD Executor, you will either be presented with the main window of the program, as shown in Figure 2.2, or the New Test Setup dialog, displayed in Figure 2.1.

Figure 2.1. New Test Setup



This dialog will be displayed when you start KD Executor for the very first time, or if you have specified not to load the last test setup on start-up. You can choose whether to create a new test setup, or open an already existing one.

Figure 2.2. KD Executor Main Window



The biggest part of the KD Executor main window is the test suite table. This is where you have all test suites, test cases and test runs gathered in one table.

You also have the result table, hidden behing the "Result" tab. This is a table similar to the test suite table. It contains all performed test runs, ordered and collected by date and time the test execution took place.

In the bottom of the main window, we have the tabs showing more details about the test runs. The "Detailed Results" tab shows more information about a certain test result. It displays the different test aspects, and whether the test case failed or passed each one of them.

The tab labeled KD Executor Errors will display any internal application error that might occur during execution of a test case. This is to avoid the test run being locked up by an error dialog.

The Stdout and Stderr shows just that: the standard output and standard error data from the tested application, which can be good to have when it comes to debugging.

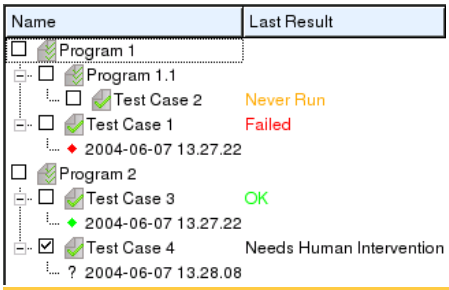
Test Suite Table

This is the place in the program where all the action is going on. You have all information about test suites and tests gathered in a tree view, and the ability to add, edit and remove test suites, tests and test runs.

The information is, as mentioned, displayed in a tree view, where the test suite is the main parent. It can then have other test suites or test cases as children. Each test case also has all of its test runs as children. Of course, it is possible to have more than one test suite in the test suite table. All this is displayed in Figure 2.3.

As visible, the last result of a test case can have four different values - OK, Failed, Needs Human Intervention or Never run.

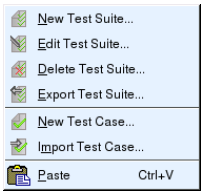
Figure 2.3. The Test Suite Table



Right-clicking in the Test Suite Table will display a menu with two options: New Test Suite and Import Test Suite. The first option will bring up the Test Suite Dialog, and the second option will display a file dialog, letting you choose a test suite file to import.

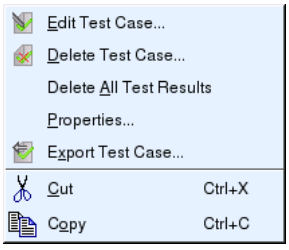
Right-clicking on any of the items, will show a menu with options for that specific item:

Test Suite item



New Test Suite...	Will display the Test Suite Dialog, giving you the option to create a new test suite. The newly created test suite will be added as a child to the right-clicked test suite.
Edit Test Suite...	Shows the test suite dialog with all information for the test suite entered into the different fields. The test suite dialog is explained further in the section Test Suite Dialog.
Delete Test Suite...	Removes the complete test suite, along with all test and test results belonging to it.
Export Test Suite...	Lets you export the test suite for future use.
New Test...	Will bring up the test dialog to create a new test. The dialog is explained in more detail in section Test Dialog.
Import Test Case...	Presents you with a dialog to import a previously exported test case.
Paste	Pastes the previously cut or copied test case into this test suite.

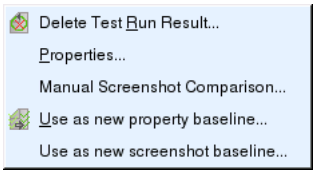
Test Case Item



Edit Test Case...	Brings up the test dialog with all information about this test case entered in the fields. The test dialog is explained in more detail in Test Dialog.
Delete Test Case...	Deletes the test case along with all its test runs.
Delete All Test Results	Delete all the test results for this test case.
Properties...	Shows the property editor with information about this test case entered in the different fields. Please see Property Editor for a more detailed description of the property editor.

Export Test Case...	Lets you export the test case for future importing.
Cut	Cuts this test case for pasting in a test suite.
Copy	Copies this test case for pasting in a test suite.

Test Run Item

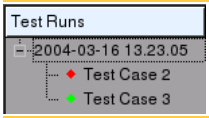


Delete Test Run Result...	Removes this test run result.
Properties...	Shows the property editor, with the properties and values from this particular test run.
Manual Screenshot Comparison...	Will display the Manual Image Compare dialog, giving you the option to look at every undetermined screenshot, and either fail it, accept it, or leave it undetermined. This menu item is only available if manual image comparison has been enabled for this test case and as long as there are undetermined screenshots.
Use as new property baseline...	Sets the values of this test run to be the new baseline in the following test runs, which means that all property values gathered during the following test runs will be compared to the property values of this test run.
Use as new screenshot baseline...	The screenshots taken during the last playback will be used as the correct screenshots in future playbacks.

Test Run Table

The test run table contains all test runs that have been performed during the current program session, stored in a tree view, and ordered in ascending time order.

Figure 2.4. The Test Run Table

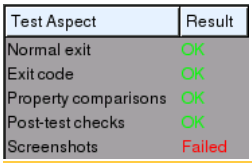


Right-clicking one of the dates presents you with a menu with two options: Delete Test Run, or Export to HTML. Exporting the test run to HTML will give you a report of all test results in that particular test run, with detailed reasons for it failing, if that occurred. You can view this HTML file in a web browser and, for example, publish it on your Intranet.

Under each date there are one or more test runs. Right-clicking one of them brings up the same context menu as in the Test Run Item section above.

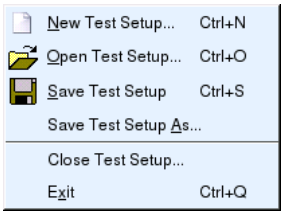
When you select one of the test results in this table or in the test suite table, there will be a more detailed report in the bottom of the main window. How this may look like is shown in Figure 2.5.

Figure 2.5. Detailed Results Tab



Menu and Toolbar

File menu



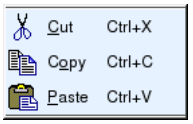
New Test Setup...

Will create a new, empty test setup. Asks you if you want to save your previous test setup, if you have modified it without saving.

Toolbar button: 

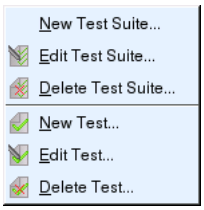
Open Test Setup...	Brings up the Open Test Setup dialog, giving you the option to open an already saved test setup. Toolbar button: 
Save Test Setup	Brings up the Save Test Setup dialog which lets you save your test setup if it is the first time you save it, or write the current test setup to the opened file. Toolbar button: 
Save Test Setup As...	Brings up the Save Test Setup dialog, which lets you save your test setup for the first time, or with a new file name.
Close Test Setup...	Will close the currently open test setup.
Exit	Shuts down the program.

Edit menu



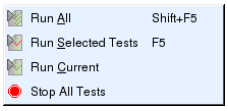
Cut	Cuts out a test case. This option is only enabled if there is a test case selected. Toolbar button: 
Copy	Copies a test case. This is only enabled if there is a test case selected. Toolbar button: 
Paste	Pastes a previously copied or cut test case into a test suite. A test suite needs to be selected to have this option enabled. Toolbar button: 

Tests menu



New Test Suite...	This brings up the test suite dialog, explained closer in Test Suite Dialog. This option is only enabled as long as there is no object selected in the Test Suite Table.
Edit Test Suite...	Brings up the same dialog as mentioned above, but in edit mode in this case. You need to select a test suite to be able to select this option.
Delete Test Suite...	Deletes a complete test suite, with all associated tests and test results. You will need to select a test suite for this action to be enabled.
New Test...	This brings up the test dialog, explained closer in Test Dialog. This option is only enabled when there is a test suite selected in the Test Suite Table.
Edit Test...	This brings up the same test dialog as above, but in edit mode instead. You will have to select a test to enable this option.
Delete Test...	Deletes a test and all its associated test runs. Naturally, a test has to be selected to enable this action.

Run menu



Run All	This option executes all test cases in all test suites. Toolbar button: 
Run Selected Tests	This option only executes the test cases that are checked. Toolbar button: 
Run Current	This option only executes the currently marked test case.

	For this option to be enabled, you will have to have a test case selected. Toolbar button: 
Stop All Tests	Will stop execution of all test cases.

Extra menu

Options...	This will bring up the Options dialog, explained in more detail in Options Dialog.

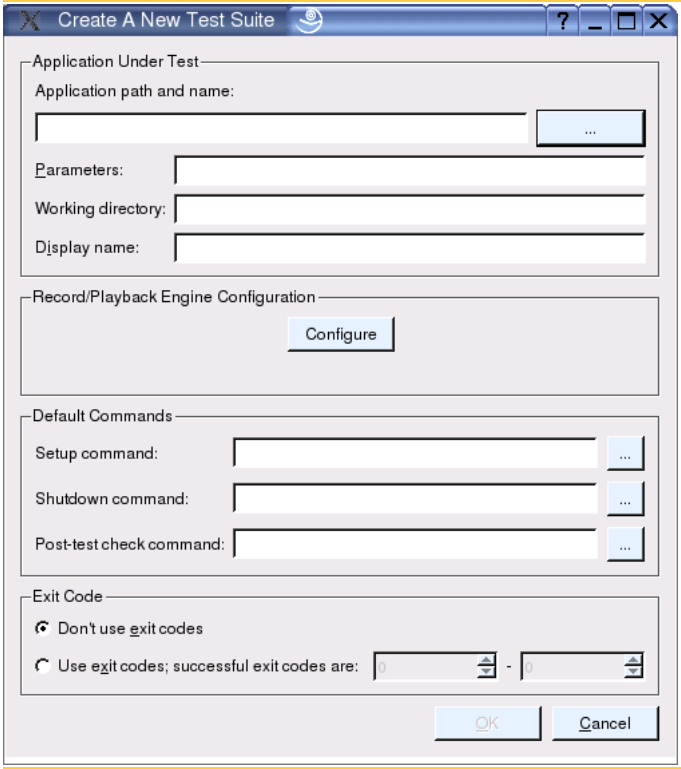
Help menu

Contents...	Displays the manual to KD Executor.
About...	Will show the About dialog.

Test Suite Dialog

The Test Suite Dialog is shown when you create a new test suite or edit an already existing one. It contains general information about which application to test, and settings for this application and the test suite. Each component in the dialog is explained in more detail further down.

Figure 2.6. Test Suite Dialog



Application path and name	This field should contain the path to the application you want to test, including the actual binary. Instead of filling this in manually you can use the "..." button to select the application. Please notice that you will not be able to change this setting once a test suite has been created (as doing so would invalidate your test cases). Also notice that you will not be able to close this dialog if the application entered here is not an executable program.
Parameters	In this field, you should enter any parameters you need to pass to the application.
Working directory	Enter the working directory for the application in this field. In most cases you can leave this empty, unless you have special requirements. If you leave this field empty, the working directory of the application will be the working directory of KD Executor.
Display name	The value of this field is what is displayed as the name of the test suite in the Test table. By default, this is set to the path to the application, including the filename.

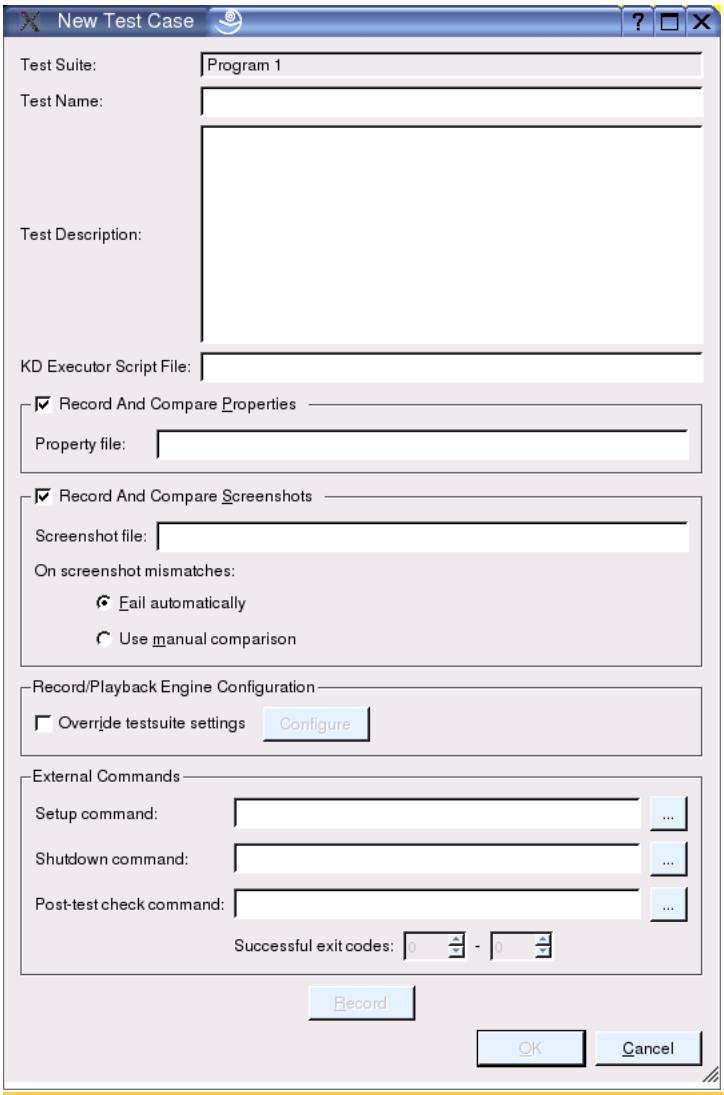
Record/Playback Engine Configuration	Click the Configure button to change the settings for the record/playback engine. Each item in the dialog that will be brought up, has a *whats this* tip associated to it, just click the question mark in the corner, and next click on the item in question.
Setup command	Here, you can specify a command that should be executed before the actual application. This way, you can execute any command you need to set up everything correctly. You can use this to e.g. set up a staging area, copying test files, fill databases, etc. The value specified here will be used as the default value for all test cases in this test suite.
Shutdown command	The shutdown command is executed after the test case is complete. Use this to clean up the staging area that was created in the setup command. The command entered will be used as the default value for all test cases in this test suite.
Post-test check command	This command can be used to check database consistency or similar to make sure that the test case worked properly. The command specified here will be used as the default command for all test cases in this test suite. Your post-test check command can pick up information about how the test case execution went by examining the environment variables `KDXTESTSETUP` (the file name of the test setup file), `KDXTESTSUITE` (the display name of the test suite; if you have not specified a display name, this is identical with the name of the application binary), `KDXTESTCASE` (the name of the test case), `KDXTESTRESULT` (the result string as displayed in the shell window, e.g. OK or FAILED), and `KDXTESTTIME` (the start date and time of the test case execution in ISO 8601 format). You can use this information to e.g. send an email message to a test engineer whenever a test case execution has failed.
Exit codes	Here you can specify whether you would like to use exit codes to determine if the test cases failed or not. If you choose to use them, you will need to specify the range for the successful exit codes. Every exit code outside this range will fail the test case.

Test Dialog

The Test Dialog is where you create/edit your test cases. It is possible to create a test case from a file that already exists, but if you choose not to, you cannot click OK before

you have recorded a script for the test case.

Figure 2.7. Test Dialog



Test Suite	This is the test suite that this test case belongs to. The value of this field cannot be changed.
Test Name	This is the name of the test case, which is shown in the Test Suite Table, as well as in the Test Run Table.

Test Description	Here you can enter a description of the test case that is shown as a tool tip in the Test table, as well as in the HTML export..
KD Executor Script File	This is the file the test case is saved to.
Record and Compare Properties	Check this checkbox if you plan to record and compare properties. If any properties mismatches during playback, the test case will be considered failed.
Record and Compare Screenshots	Check this checkbox if you would like to record and compare screenshots. If any screenshot taken during recording mismatches the screenshot taken during playback, you have two options mentioned below.
On screenshot mismatches	If you choose to record and compare screenshots, and the comparison fails, you can choose here whether you would like to have the test case fail automatically, or if you rather would like to compare the screenshots manually.
Record/Playback Engine Configuration	To use unique record/playback engine configuration for this particular test case, check the checkbox to enable the Configure button, and then change the configuration to your requirements.
Setup command	This is a command that is executed before the actual application, just as described in Test Suite Dialog.
Shutdown command	The command specified as the shutdown command is executed after the test case is complete, just like the shutdown command described in Test Suite Dialog.
Post-test check command	As described in Test Suite Dialog this command can be used to check e.g. database consistency after a test case is completed, or to send email messages about the test execution result.
Successful exit codes	Here, you should specify which exit codes that should be considered successful for the post-test check command.

Property Editor

When you choose to view the properties of a test case or a test result, the property editor is displayed. It comes in two different versions.

Figure 2.8. Property Editor (Test Case)

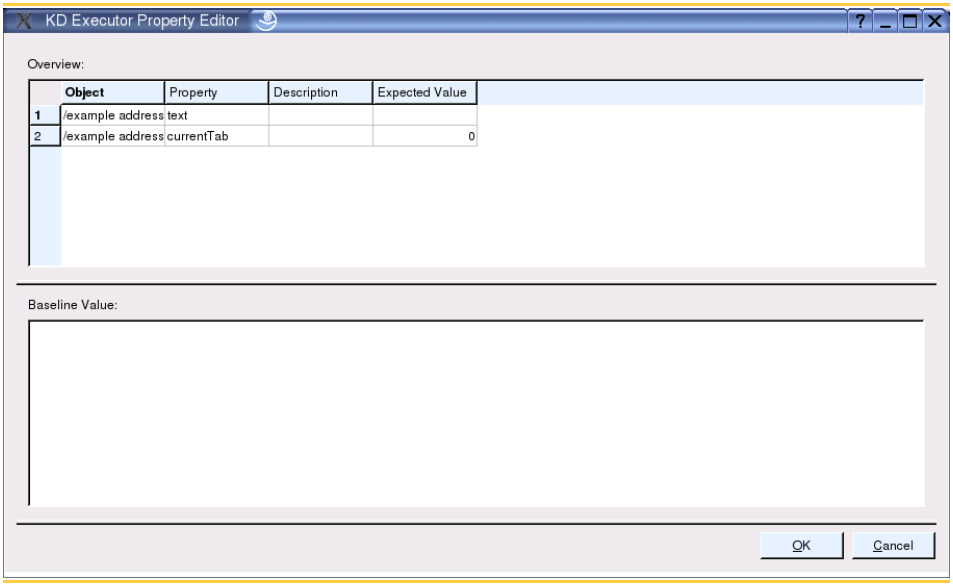
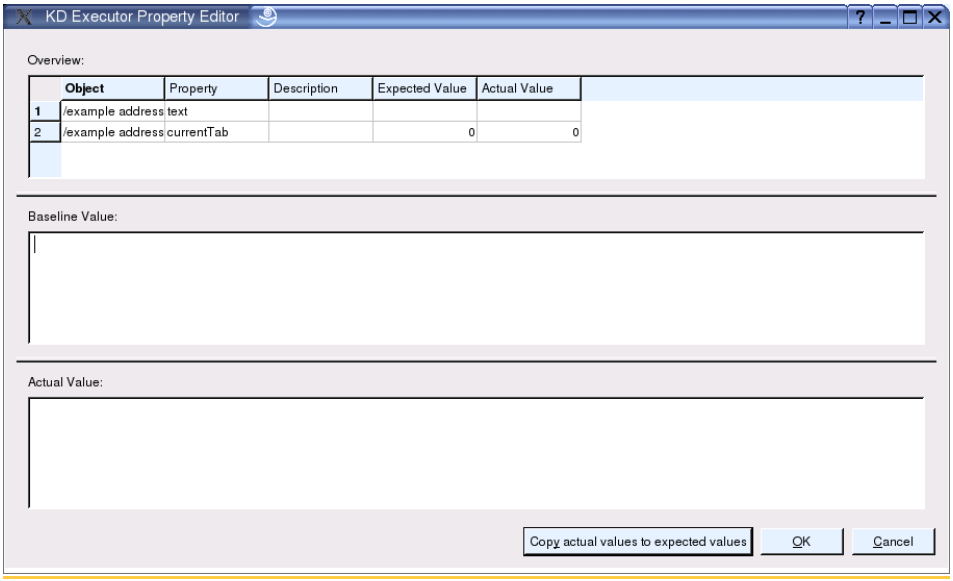


Figure 2.9. Property Editor (Test Result)



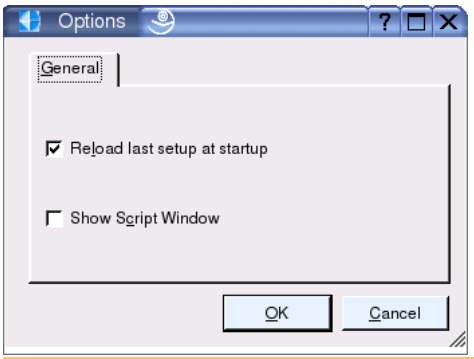
Listed in the Property Editor are all properties picked during the script recording. The only differences between the two dialogs is that the property editor for a test result, also shows the actual value of the property, captured during playback. There is also a button "Copy actual values to expected values". Clicking this button will overwrite the expected values, also called the baseline, with the actual values.

Discrepancies Between the Number of Baseline Values and Actual Values

If you include a prerecorded script piece in your test case, and change that script piece later to the extent that a different number of properties is recorded now, there will be a discrepancy between the number of properties that were recorded during test recording (the so-called *baseline*) and the actual values recorded during test playback. A warning message box will be issued in this case, and if you copy the actual values to the expected values as described above, the missing values are filled in, and surplus properties are deleted.

Options Dialog

Figure 2.10. Options Dialog

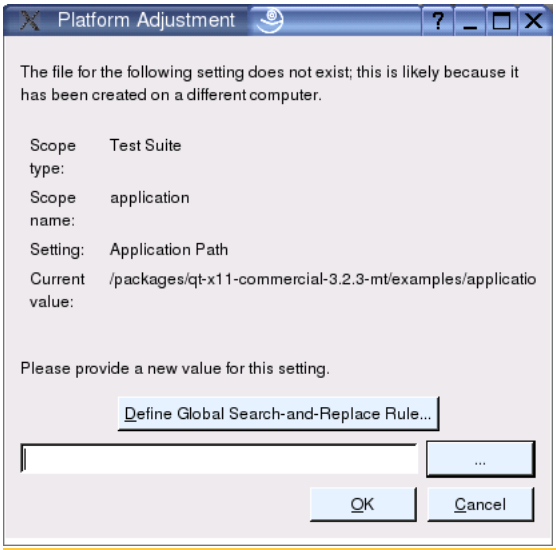


Reload last setup at startup	Having this option checked will load your last test setup the next time you launch KD Executor.
Show Script Window	If you check this option, a part of the main window will become occupied by a script window. This window will contain the actual XML script of the currently selected test case. This is intended for advanced users that want to make manual adjustments to a test script while working with tests. Normally, you do not need this.

Transfer Test Setup

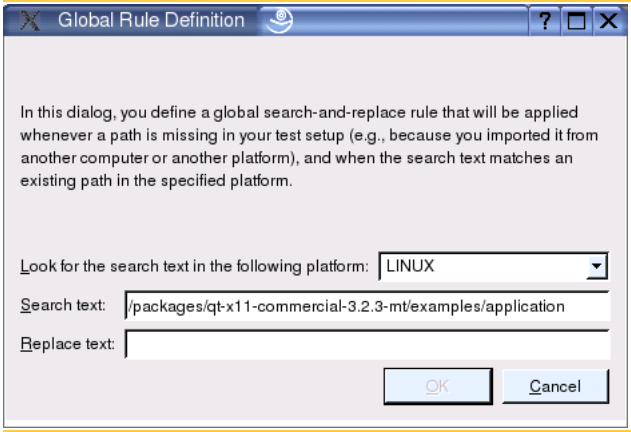
Loading a test setup that has been created on another computer into KD Executor, will require you to specify where the application under test is located, and where all the different scripts are located. This is handled in the Platform Adjustment dialog.

Figure 2.11. Platform Adjustment



You will get a question like this for every setting in the test suite, except for those settings where all platform dependent values are empty. If you have several test suites and/or test cases, this will bring up several questions. In this situation, you can use the Global Rule Definition to define search-and-replace rules that should be used when importing the test suites and test cases.

Figure 2.12. Global Rule Definition



Using this dialog, you can specify that all settings looking a certain way should be replaced with something else. This is very convenient when transferring several test cases that are all in the same directory, as you replace the previous directory with the new one.

Note

The setting is only replaced if it can find a file using the replacement value.

If the test scripts you are transferring compares screenshots during playback, you will manually have to transfer the recorded screenshots. These are located in the same directory as the test scripts and the test setup. This will be improved in future releases.

Chapter 3. The KD Executor Sidebar

While recording, a number of special KD Executor actions are available. These actions are available from a side bar which is shown together with the application being recorded (see Figure 3.1). In addition, it is possible to bind these functions to keys from the `Key Bindings` tab in the configuration dialog as can be seen in Figure 3.2.

Figure 3.1. KD Executor side bar

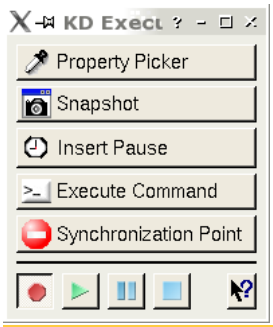
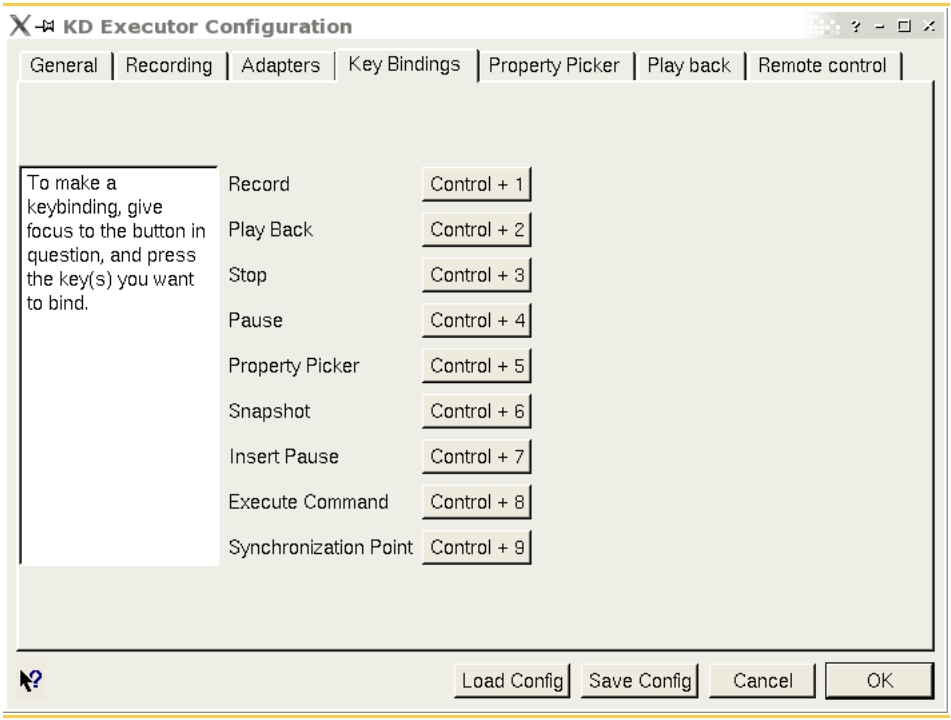


Figure 3.2. Configuring Keybindings



To make a key binding simply select the button for the action you are interested in, and press the key combination of interest.

The following list contains description for each of the available recording features.

Record	Start recording.
Playback	Start play back.
Stop	Stop recording or playback.
Pause	Pause recording or playback.
Show Property Picker	If you use KD Executor for regression testing then you may use the property picker to dump the content of widget properties. See Chapter 4, *Printing Interface Properties* for details on the property picker.
Snapshot	Asks you to point with the mouse on a dialog or a widget, and then makes a screen shut of that dialog or widget.
Insert Pause	If you specify *compressed time>* in the configuration dialog, then KD Executor will playback your script as fast as possible. If, however, you want it to stop for a short while

	at a certain point then this action will let you specify an amount of time to stop. You may wish to wait for a short period from time to time, to avoid that KD Executor delivers events to say a list box before it ever contains any elements.
Run External Command	Executes an external command on your system
Synchronization Point	Similar to *Insert Pause* above, using this command you may make KD Executor wait with delivering events until a certain condition has been meet - say the list box got any items. This of course, is not only useful if KD Executor plays back as fast as possible, but also if the backend filling the list box, might be slow (the items may for example come in over a network connection or come from a database).

If KD Executor is compiled into your binary (see Section , “KD Executor—Simple Source Code Modification”), then the recorder dialog may unfortunately be blocked by any modal dialog of the application. In these situation you have to use keybindings instead.

If you are linking KD Executor into your application, then you may make the key binding using `KDExecutorFeatures::setAction()`, as can be seen in Figure 3.3.

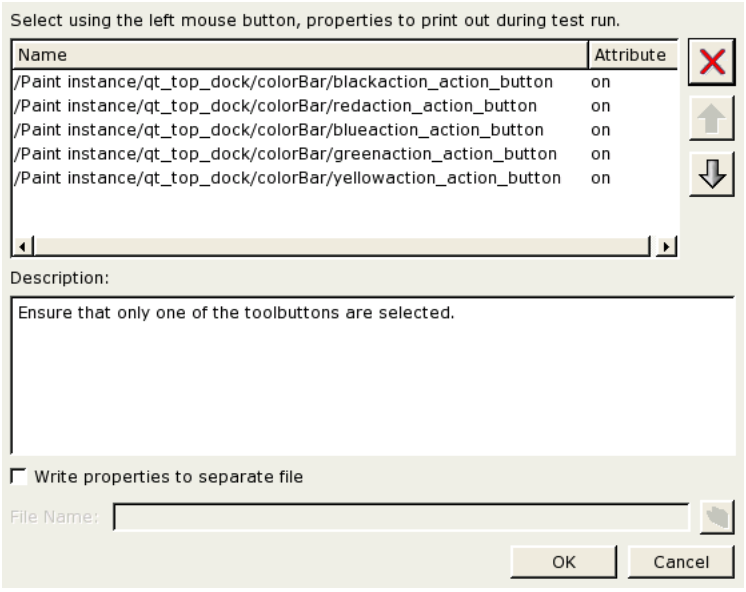
Figure 3.3. Example Of Specifying Key Bindings For Special Recording Features

```
int main( int argc, char** argv ) {
    QApplication app( int argc, char** argv );
    KDExecutorFeatures::instance()->setAction( KDExecutorFeatures::Stop,
          Qt::SHIFT+Qt::Key_F8 );
    KDExecutorFeatures::instance()->setAction( KDExecutorFeatures::Pause,
          Qt::SHIFT+Qt::Key_F9 );
    KDExecutorConfig* config = new KDExecutorConfig( argc, argv );
    config->exec();
    MainWindow* w = new MainWindow( 0 /*parent*/ );
    config->start( w );
    int ret = app.exec();
    config->stop();
    return ret;
}
```

Chapter 4. Printing Interface Properties

If you use KD Executor for regression testing, you might be interested in being able to tell whether a list box contains a given item, whether a check box is enabled etc. In other words, you might be interested in getting the current state of Qt widgets.

Selecting `Show Property Picker` in the recorder dialog, will bring up a dialog that looks like this:



The property picker allows you to click on widgets in your interface, and choose properties to be printed during test execution. The properties are Qt properties as described in the Qt reference documentation for each widget. In addition extra properties might be developed as described in Chapter 8, *Developing Additional Properties*.

Often, you are interested in picking the same properties several times for specific widgets, and to avoid having to do this manually you can save the property pick for later use.

To do this, the first time, go to the property picker, select the properties of the widgets you want, and also check the check box `Write properties to separate file`. Following times, press the `Play` button in the side bar and choose the script file you saved to in the step above. This will play back the property pick.

Chapter 5. Using the KD Executor Core Engine

If you plan to use KD Executor for automatic testing, then the KD Executor shell as described in the previous chapter might be all you need, and you can safely skip this chapter. On the other hand, if you anticipate to compile KD Executor into your application for one of the reasons described in the introduction, or you plan on using the `kdexecutor` binary rather than the `kdxshell` then please read on.

If you do not use the KD Executor shell, then there are two alternative ways of using KD Executor:

- *Preloading* — The by far easiest and normal way of using KD Executor. In this setup, your source code *do not need* to be modified. This is the way used when using KD Executor from the shell.
- *Source code modification* — It is possible to compile KD Executor into your binary. Normally there are two situation where you may wish to do so. Either you need to limit the usage of your software (e.g. for a playback-only demo version), or you want to build KD Executor into an already tailored C++ test environment.

Preloading KD Executor

KD Executor may be used without any modify to your Qt application. Simply execute your application using the `kdexecutor` binary that you will find in `$KDEXECUTORDIR/bin`.

The following is an example of using KD Executor on Linux:

```
$KDEXECUTORDIR/bin/kdexecutor -record $QTDIR/examples/demo/demo
          -style platinum
```

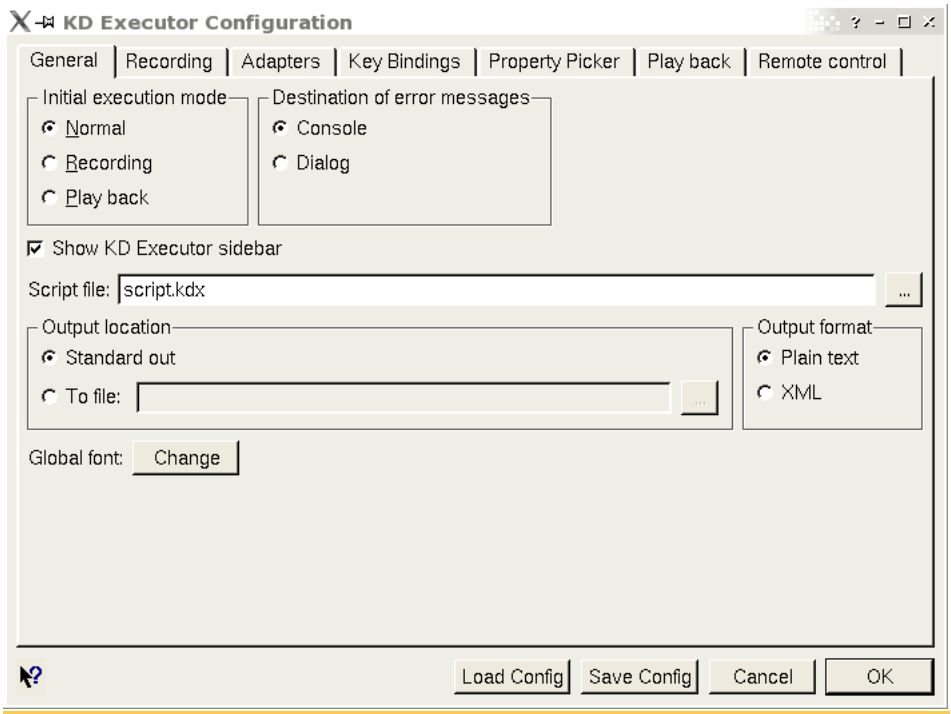
On windows you may simply click on the KD Executor icon, and specify the command to executor in the dialog that shows up. Alternatively you may use a command line similar to the following:

```
%KDEXECUTORDIR%\bin\kdexecutor -record %QTDIR%\examples\demo\demo.exe
-style platinum
```

When typing in the commands above, no line breaks should be typed, they are merely for ease of readability.

When KD Executor is started the configuration dialog in Figure 5.1 will be shown. In this dialog you can configure execution mode (recording or play back), plus a number of options regarding the configuration. Pressing the question mark in the lower left corner will allow you to get extensive help for each option.

Figure 5.1. KD Executor Configuration Dialog



The configurations in the dialog can be saved to a file by pressing the Save Config button at the bottom of the dialog. Later you may load a previous saved configuration using the Load Config button, or by specifying the file which the configuration was saved to using the command line argument `-configfile`

KD Executor understands a number of command line options, invoke the `kdexecutor` binary using `-help`, to see the list.

KD Executor—Simple Source Code Modification

As an alternative to using KD Executor as a wrapper application around your own application, you may compile KD Executor into your application. There are a number of reasons to why you may wish to do that:

- Your system does not support pre-loading (This is currently known to work on Microsoft Windows, Mac OS X, and Linux, and likely to work on any ELF based UNIX system)
- You want to ship a binary that only supports playing back a predefined script.
- You want to ship a binary with KD Executor support, but do not want to trouble your users with starting your binary through another one.
- You want to compile your application using static libraries (pre loading requires KD Executor is a dynamic library)

If none of the items above matters for you, then you may skip the rest of this chapter, and continue on to Chapter 3, *The KD Executor Sidebar*.

There are two different ways you can modify your application for use with KD Executor. Either you may uses a predefined dialog for letting the user control KD Executor [1], or you may take full control yourself. The latter is of course more work for you as a developer, but it allows you to integrate KD Executor seamlessly into your application. We will in the following look at each of the possibilities in turn.

The simplest way of integrating KD Executor into your application is to use the class `KDExecutorConfig` as can be seen in the following code snippet.

```
int main( int argc, char** argv ) {
    QApplication app( int argc, char** argv );
    KDExecutorConfig* config = new KDExecutorConfig( argc, argv );
    config->exec();
    MainWindow* w = new MainWindow( 0 /*parent*/ );
    config->start( w );
    int ret = app.exec();
    config->stop();
    return ret;
}
```

The changes required by KD Executor are typeset in boldface. Calling `config->exec()` will bring up a configuration dialog as shown in Figure 5.1.

After having created the main window of the application you must call `config->start()`, which will start recording or playback depending on the configuration in the configuration dialog. After `app.exec()` returns, you must call `config->stop()`, which will stop recording in case the current execution mode is recording.

When creating the `KDExecutorConfig` instance, you pass the command line arguments *argc* and *argv*. KD Executor interprets certain command line arguments as de-

1This is the dialog if Figure 5.1

scribed in the previous section. When KD Executor finds one of the command line argument it knows about, KD Executor will remove it from `argc`/`argv`.

In case you do not want KD Executor to take over the *namespace* for options, then you may specify a letter combination to be used in front of the KD Executor options. For example `kdx`, which would give you options `-kdxhelp`, `-kdxrecord` etc. To do this specify a forth argument to the constructor of KD Executor.

All the other features configurable in the configuration dialog can be controlled from source code by the class `KDExecutorFeatures`. Please see the reference documentation for further details on that class.

KD Executor—Full Control

To get full control over KD Executor, you may create the instances of the classes `Recorder` and `PlayBack` yourself. You do, however, seldom need that much control. An example where you might, however, want this control is for a demo version of your application, where you do not want the user to be capable of fiddling with all the options of KD Executor.

When you take control of the creation of `Recorder` and `PlayBack` objects yourself, then you may still use `KDExecutorConfig`; simply specify false for the third argument to the constructor. [2]

When creating the `Recorder` and `PlayBack` yourself, you must remember to create the instances before any widget is created. This does in general mean that you need to create them just after you have created the `QApplication` instance.

Figure 5.2 is an example of an application which is enabled for recording, while Figure 5.3 shows an application enhanced for play back. To make one application capable of both recording and playing back, you do of course need to have the two program listings combined with code for checking whether you want to record or play back.

Figure 5.2. An Example of an Application Enhanced for Recording

```
int main( int argc, char** argv ) {
  QApplication app( int argc, char** argv );
  Recorder* recorder = new Recorder( 0 /*parent*/);
  MainWindow* mainW = new MainWindow( 0 /*parent*/ );
  recorder->start( "scriptFile" );
  int ret = app.exec();
  recorder->stop();
  return ret;
}
```

2An alternative is to ask `KDExecutorConfig` to create the instances for you, and when needed get a pointer to the actual `Recorder` and `PlayBack` using the methods `KDExecutorConfig::recorder()`, and `KDExecutorConfig::playback()`.

Figure 5.3. An Example of an Application Enhanced for Playback

```
int main( int arg, char** argv ) {
  QApplication app( argc, argv );
  PlayBack* player = new PlayBack( 0 /* parent */ );
  MainWindow* mainW = new MainWindow( 0 /* parent */ );
  player->run( "scriptFile" );
  return app->exec();
}
```

The argument to the constructors of `Recorder` and `PlayBack` is the file in which the script is located. KD Executor may also be used for controlling an application remotely. In this situation, KD Executor scripts are not written to or read from a file, but instead to a socket on which the remote application is listening. In this situation a different set of constructors for the `Record` and `PlayBack` is used.

Part II. Enhancing KD Executor

Table of Contents

6. Getting the Best Results with KD Executor .....
Surviving Source Code Changes 36
7. Adapters .....
Developing Your Own Adapters 39
An Example Adapter 40
Putting the Adapter to Work 43
8. Developing Additional Properties .....
An example property extension 44
Putting the Property Extension to work 48
9. Frequently Asked Questions .....
10. The Format Of The Script Files .....
Low-level Events 51
High-level events 54
KD Executor Events 54

Chapter 6. Getting the Best Results with KD Executor

In order to get the best results with KD Executor, it is important that you name your objects—best result here refers to KD Executor's chances at successfully playing back a script recorded with an older revision of your source code, even after you have made changes to the code and to the layout of your application windows and dialogs.

Objects are named by giving a string to the last part of the constructor, just after the parent argument. For example when creating a `QLabel`: `QLabel* title = new QLabel( tr("Title"), this, "title_label")`[3]

Don't despair if you have a 1.000.000 lines program without any names for objects, and on which you want to use KD Executor. KD Executor can very well get along without—you may add names as you go on, and not all objects need to be given names for optimal playback, so please read on.

To understand when and why the names are needed, lets get behind the scenes of KD Executor execution. Basically, KD Executor listens to events in your application during recording[4]. During playback, it emits synthetic events to the widgets so that the application believes that a user is interacting with it. The order of playback and the widgets that get the events are of course the same during playback, as during recording.

The tricky part in all this is to identify the widgets to send the events to during playback. During recording, we only know the widget by having a pointer to it—this pointer is of course invalid during playback. But still, we need to write something into the script file which later can be translated into a pointer to a widget.

In order to achieve that, we use the names given to the widgets plus the parent-child relationship build during widget construction.

Let's go through Figure 6.1 below line by line, and see which name each object gets.

Figure 6.1. Code example for object naming

```
1 class MyDialog :public QDialog {
    MyDialog( const char* name ) :QDialog( 0, name ) {
      ...
      QWidget* top = new QWidget( this, "container" );
5     ...
      QLabel* title = new QLabel( tr("Cd Title:"),top, "title" );
      QLabel* artist = new QLabel( tr("Artist:"), top /* no name */);
      QLabel* id = new QLabel( tr("Id:"), top /* no name */);
```

3Due to technical reasons, you cannot specify the name by calling `QObject::setName()` after construction of the object—doing so will have no effect to KD Executor.
4mouse events, key events, focus events, etc.

```
    }
10  ...
}
```

The dialog does not have a parent (we create it in line 2 with 0 as the parent pointer in the QDialog constructor), the name that the object gets in KD Executor is therefore simply the name given as argument to the constructor. If we had created an instance of this object like this: `new MyDialog( "aDialog" )`, then that instance would have been named `/aDialog`.

The QWidget container we create in line 4 is a direct child of the dialog, and will therefore be named `/aDialog/container`, likewise the label created in line 6 is a child of the container widget, and will therefore be named `/aDialog/container/title`

The label in line 7 does, however, not have any name assigned to it, so its name (seen from Qt) is simply `unnamed`, therefore it will be named `aDialog/container/unnamed` in KD Executor.

Finally the label created in line 8 doesn't have a name either, but since it is a sibling to the label from line 7, we need to append a number to get a unique name for it; it would be named `/aDialog/container/unnamed1`.

Surviving Source Code Changes

During playback, KD Executor will use as much of the name as it needs to find a unique widget, this has the advantage that even if part of the parent/child tree changes names, KD Executor will still find the widget. As an example, if you change the name of the dialog from `aDialog` to `myDialog`, then all events might still be delivered to the widgets in the dialog.[5]

This may not sound like a big issue, but it most certainly is if you have not named all your widgets. Imagine that the dialog had been created without a name; then it would have been named `/unnamed` or perhaps even `/unnamed42` in case 42 other widgets without parents had been created before it. Now if you add e.g. a QTimer (which also inherits QObject, and thus claims a name) before your dialog, then the name of the dialog will change to `/unnamed43`, and had KD Executor not been clever about searching for widgets you would have to re-record all your scripts.

The moral of all this is: name as many widgets in your application as needed to get all objects to which events are delivered uniquely named. Thus in the example above, if you always only create one instance of the `MyDialog` class, then you might not need bothering about naming it, as its children are likely to contain enough names on their own to be unique.

5Whether your script will still work, depends on whether the rest of the name is unique, i.e., whether any other widgets end in `container/title`, for example.

There is one important special case to naming, namely context sensitive menus - these often are created without a parent, so you should always remember to name these.

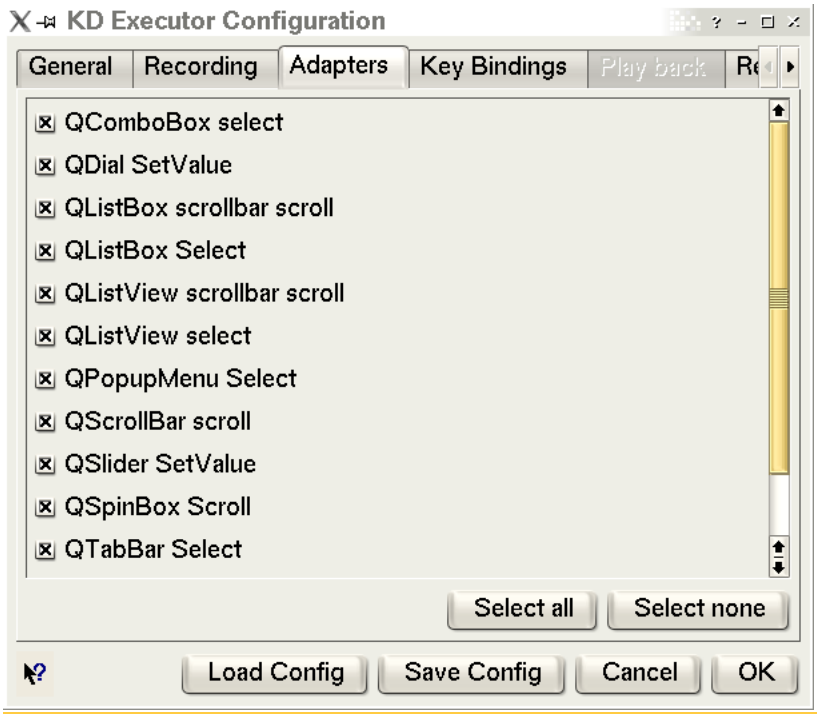
On a side note, if you write scripts by hand, then you do not need to name the complete name in the script file, you may cut off as much of the beginning of the name as you want, under the condition that it will still be unique. Thus in the example above, you could simply write `title` rather than `/aDialog/container/title`.

Chapter 7. Adapters

As described in the introduction, KD Executor takes care of recording the intention rather than the action in a number of specific cases. This is done using adapters.

An adapter can be a pretty difficult thing to write, since it must ensure that the played back application sees no difference compared to what it saw during recording. In other words, the adapter must be written in a way that ensures that the widget it adapts emits the same signals during playback as they did during recording. Despite how much care are taken when writing adapters, they may still break playback in rare situations. Therefore, KD Executor offers the person recording a script to enable or disable individual adapters. This is done in the `Adapters` tab in the configuration dialog shown at startup, as can be seen from Figure 7.1

Figure 7.1. Adapter Configuration



If you link KD Executor into your application, you may get even further control over adapters using `KDExecutorFeatures::setAdapterEnabled()`. With this method, you can disable a given adapter for all widgets simultaneously, or for individual widgets.

Two convenience methods exist for enabling or disabling all adapters for all widgets

simultaneously, namely `KDExecutorConfig::enableAllAdapters()` and `KDExecutorConfig::disableAllAdapters()`.

Developing Your Own Adapters

KD Executor is shipped with a number of adapters, and it is possible to create new adapters yourself. You may wish to do that either because you want KD Executor to handle predefined Qt widgets in a specific way, or because you have created a new widget yourself.

Before we get into the technicalities about developing an adapter, lets first revisit why you might need to develop an adapter.

Every application has some widgets that are special to the application domain, let that be an *oil pump widget*, a *compass widget*, a *connection editor widget* or a *geographical map widget*. These widget may handle say mouse press inside them different depending on the location inside the widget, just like the tabbar in Chapter 1, *Overview*.

If say a geographical map widget is having a different size at playback time than it did during recording, then the countries on the map will be located at a different place during playback than they did during recording, and you will therefore need an adapter to ensure that the correct country was selected upon mouse clicks for example.

Now lets get to the technical part of developing an adapter. The adapter we will develop is a widget resembling a list box, simply because that widget is much simpler than say a map widget.

An adapter consists of two parts. The first part is used to recognize specific events to specific widgets at recording time. An example would be to recognize mouse press events on instances of the class QListBox. The second part is to emulate the event on play back time. For the list box example, this could be done by posting the appropriated mouse events or by calling `setSelected()` on the list box instance.[6]

During recording, the adapter will indirectly write an XML element to the script file, and during playback the adapter will recognize this element and execute whatever needs to be done. If you are not interested in using KD Executor for recording scripts, but instead want to write your own scripts by hand, then you may choose to create adapters for playback only that will understand the new tags you have invented.

Adapters are therefore separated into two classes, `PlayBackAdapter` and `RecorderAdapter`. `RecorderAdapter` inherits `PlayBackAdapter`, so when you implements a `RecorderAdapter`, you also need to implement a `PlayBackAdapter`.

Implementing an adapter consists of inheriting from either `PlayBackAdapter` or `RecorderAdapter`, and overriding a number of virtual methods.

Implementing a recording adapter requires that you override at least the following two

6Calling `setSelected()` will, however, result in certain signals not being emitted for example `selected(int index)`

methods:

`info()`	This method must return some information about the adapter. The first part of the returned `Info` struct is a name for the adapter. This name is used when enabling/disabling adapters programmatically. The second part is a short description, while the last part is a longer description. Both used in the adapter page show in Figure 7.1.
`recordingEvent()`	This is the method that looks at events, and generates an XML element describing the event. For a detailed description of this method, please refer to the reference documentation.

Implementing a play back adapter requires that you override the method `playBackEvent()`. This method takes as argument an XML element from the script file, and an object for which the *event* is meant for, and must execute the event on the object.

An Example Adapter

In the directory `examples/adapter-no-plugin/` an example of an adapter is available [7]. The implementation contains a widget which looks a lot like a list box. This widget is the one we are interested in creating an adapter for.

The widget shows a number of items above each other, but each item is not a widget on its own. That means that when the user selects an item in the list box, then it is the widget with all the items that actually gets the mouse press event, not the individual items. This again means that if the font size on play back time is different from the font size on recording time, then a different item would be located underneath the cursor, and thus a different item will be selected. To overcome this problem, we implement an adapter that listens to mouse press events on instances of the widget type in question, and when it sees one, it records the item number instead of the mouse press event.

Figure 7.2. Header file for sample adapter

```
#ifndef EXAMPLEADAPTER_H
#define EXAMPLEADAPTER_H

#include <adapter.h>

class ExampleAdapter :public RecorderAdapter {
public:
    EventReplacement recordingEvent( QObject* watched, QEvent* e,
                                     bool spontaneous, QDomDocument& doc );
    bool playBackEvent( QObject* obj,  const QDomElement& elm );
```

7 If you have purchased KD Executor, you may find more examples in the `plugin` subdirectory of the KD Executor distribution.

```
        Info info() const;
    };

    #endif /* EXAMPLEADAPTER_H */
15
```

The header file for the example adapter can be seen in Figure 7.2. The class inherits `RecorderAdapter`, and is thus both a recorder adapter and a play back adapter. For the recorder adapter part, it implements the methods `recordingEvent()`, and `info()`. For the play back part, it implements `playBackEvent()`.

Figure 7.3. Source File For Sample Adapter

```
 1 #include "exampleadapter.h"
   #include "mywidget.h"

   EventReplacement ExampleAdapter::recordingEvent( QObject* watched, QEvent* e,
 5                                                  bool spontaneous,
                                                    QDomDocument& doc )
   {
       if ( watched->inherits( "MyWidgetNoPlugin" ) &&
            e->type() == QEvent::MouseButtonPress && spontaneous ) {
10         MyWidgetNoPlugin* w = static_cast<MyWidgetNoPlugin*>(watched);
           Q_ASSERT( w );
           int index = w->itemAt( static_cast<QMouseEvent*>( e )->pos() );
           QDomElement node = doc.createElement( "MyWidgetNoPlugin::select" );
           node.setAttribute( "index", index );
15         return EventReplacement( node );
       }
       return EventReplacement( EventReplacement::NoBlock );
   }

20 bool ExampleAdapter::playBackEvent( QObject* obj,  const QDomElement& elm )
   {
       if ( elm.tagName() == "MyWidgetNoPlugin::select" ) {
           int index = elm.attribute( "index" ).toInt();
           if ( obj->inherits( "MyWidgetNoPlugin" ) ) {
25             MyWidgetNoPlugin* w = static_cast<MyWidgetNoPlugin*>( obj );
               w->setSelected( index );
           }
           return true;
       }
30     else
           return false;
   }

   RecorderAdapter::Info ExampleAdapter::info() const
35 {
       return Info( "ExampleAdapterNoPlugin",
                    QObject::tr( "Example adapter not compiled as a plugin" ),
                    QObject::tr( "Example of an adapter. Source code is available "
                                 "in examples/adapter-no-plugin" ) );
40 }

   KDEXECUTOR_ADAPTER( ExampleAdapter )
```

The implementation of the adapter can be seen in Figure 7.3.

Let us start by looking at the simplest method of the three, namely `info()`, which can be seen on lines 34-40. This method returns an `Info` struct, which contains three parts. The first part is the name of the adapter. This is the name used by `KDExecutorFeatures::setAdapterEnabled()`. This name just need to be a unique among all adapters available. The second part is the title that you e.g. can see in Figure 7.1. Finally, the third parameter is the help message that you get from the dialog in Figure 7.1, when you press the Help button, and choose the adapter in question.

The next method is `recordingEvent()` which can be seen on line 4-18. This method is the one that is invoked at recording time, and has the opportunity to return an XML element to be written to the script file instead of the default one describing the event.

The first part of the method is an *if statement*, which tests whether this event is one of the events this adapter takes care of, and if the widget is the widget type the adapter is interested in. This adapter is an adapter for the widget class `MyWidgetNoPlugin`, and the adapter is only interested in mouse press events. In addition, it is only interested in spontaneous events [8].

On line 12, we call the method `itemAt()`, to get the index of the item at the position of the cursor. Remember our reason for implementing an adapter is to remember the index of the item selected rather than the mouse press itself.

Having mapped the mouse press to an index, all we need to do is to create an XML element, which is done on line 13. The name of the element is the argument to the `createElement()` method. The name is one we invented ourself, but it must of course be the same as the name we match against on line 22. Finally we make the id an attribute of the element on line 14, and returns the element on line 15.

In case the event was an event not to be handled by this adapter, the adapter returns an *empty* `EventReplacement` telling KD Executor to try other adapters. This can be seen on line 17.

The last part of our code is the method `playBackEvent`, which takes care of handling the event at play back time. The first part of the method tests to see if the XML node is one of those we created on line 13, i.e. one with tag `MyWidgetNoPlugin::select`. If that is the case, we will read the `index` attribute we wrote on line 14. Having read that index, we simply invoke the method `setSelected()`, which is a method of the widget we are interested in.

Notice that the above adapter has one flaw, namely that at playback time it will invoke the method `setSelected()` rather than post mouse events to the adapter. Posting events to the adapter is slightly more complicated, but has the advantage that the code of the widget see mouse press and mouse release events, plus appropriate signals are emitted. To see real life industrial strength adapters, please consult the plugin subdirectory of your KD Executor installation.

8A spontaneous event is one which comes from the operating system in contrast to one send by the user using `QApplication::sendEvent()` or `QApplication::postEvent()`. It is only in very rare situations that you should care about non-spontaneous events.

With the above in place we now have managed to map a mouse press event on the widget at recording time to a `setSelected()` method call on play back time. Now the widget still works even when the font size changes.

The final pice of code yet unexplained is line 42, this is required if the adapter is compiled as a dynamic loadable plug-in. We will discuss that in detail in the next section; for now just accept that this line should always be located in the implementation file for adapters. Notice that the name inside the parentheses is the name of the class we just defined.

Putting the Adapter to Work

Once the adapter class has been created, you must either compile it as a plug-in for KD Executor, or if you compile KD Executor into your application, you may call `KDExecutorFeatures::addAdapter()` with an instance of the adapter [9].

Compiling a plugin for KD Executor requires that you compile the code as a library with the preprocessor define `QT_PLUGIN` set. (This can be done by adding a line saying `CONFIG+=plugin` to your QMake project file. You must also add a define for `KDEXECUTOR_DLL`, which in QMake syntax would be the line saying `DEFINES += KDEXECUTOR_DLL`. This library needs to be put in `$KDEXECUTORDIR/lib/plugin`

An example of an adapter compiled as a plugin can be seen in `examples/adapter-plugin/adapter`, and an example of one compiled into the application can be seen in `exmaples/adapter-no-plugin`

9Normally you would put code like the following in your main function: `KDExecutorFeatures::instance()->addAdapter( new MyAdapter() );`

Chapter 8. Developing Additional Properties

Qt widgets sometimes do not have a property for the information you want to have a closer look at. An example of such a *missing* property is a property containing the elements of list box.

It is, however, possible for you to add your own properties to Qt widgets - at least as far as KD Executor is concerned. [10] To do so, you must inherit from the class `PropertyExtension`, and implement the following three virtual methods:

`properties()` — This method expects a pointer to a QWidget, and it must then return the list of property names it offer for this widget. For each item, you also specify a category which determines in which submenu of the property picker the item will show up.

`value()` — This method expects a pointer to a widget, the name of a property, and a `RecorderInfo` struct (described in the next bullet item). It must return a QVariant representing the value of the property at the given position in the given widget. The Value method is overloaded to allow to return information about a number of properties, this is described a bit later.

`recorderInfo()` — In Chapter 7, *Adapters*, we talked about adapters. The important thing was to make the script play back correctly with different settings (different geometry, different font, or even a different language). The same problem exists for property extensions. Therefore it is possible for the property extension to bring information about the *situation* from recording time (where the user specify the property), to play back time (where the property is printed). Such information could be the ID of the tab under mouse, in the example of a QTabBar. The method `recorderInfo()`, must return a map with such information. The map is given as an argument to the `value()` method described in the bullet item above.

An example property extension

In the subdirectory `examples/properties-no-plugin/` an example of property ex-

10KD Executor itself already implements additional properties for a number of Qt widgets.

tensions is available [11]. The example consists of an application showing most of the widgets available with Qt, plus an implementation of a property extension.

Figure 8.1. Header file for sample property extension

```
#ifndef MYPROPERTYEXTENSION_H
#define MYPROPERTYEXTENSION_H

#include <kdabplugin.h>
#include <propertyextension.h>
#include <qtabbar.h>

class MyPropertyExtension: public PropertyExtension
{
public:
    virtual PropertyExtList properties( QWidget* widget );
    virtual RecorderInfo recorderInfo( QWidget* object,
                                       const QString& propertyName,
                                       const QPoint& pos );
    virtual PropertyValueList value( QWidget* widget,
                                     const QString& propertyName,
                                     const RecorderInfo& recorderInfo );
    virtual QVariant value( QWidget* widget,
                            const QString propertyName,
                            const RecorderInfo& recorderInfo,
                            bool& valid );

protected:
    void dumpProperties( PropertyValueList& result, QWidget* obj );
    void dumpAllProperties( PropertyValueList& result, QWidget* dialog );
};


#endif /* MYPROPERTYEXTENSION_H */

```

The header file for the example property extensions can be seen in Figure 8.1. The class inherits `PropertyExtension` and implements the three methods described above.

Figure 8.2. Implementation file for sample property extension

```
#include "mypropertyextension.h"
#include <KDStream.h>
#include <qmetaobject.h>
#include <qvariant.h>
#include <kdexecutorutil.h>
#include <qobjectlist.h>
#include <qobject.h>
#include <qapplication.h>
#include <qwidgetlist.h>

PropertyExtList MyPropertyExtension::properties( QWidget* widget )
{
    PropertyExtList list;
    if ( widget->inherits( "QTabBar" ) ) {
        list << PropertyItem( "Example Extensions - No Plugin",
```

11Extra plugins may be found in the `src/plugins` subdirectory of the KD Executor distribution.

```
                                "Current Tab Name" )
                << PropertyItem( "Example Extensions - No Plugin",
                                "Name of Tab at Point" );
        }
20      list << PropertyItem( "Example Extensions - No Plugin",
                              "All Properties of Widget" )
             << PropertyItem( "Example Extensions - No Plugin",
                              "All Properties of Dialog" )
             << PropertyItem( "Example Extensions - No Plugin",
25                            "List of Top Level Windows" );
        return list;
    }

    RecorderInfo MyPropertyExtension::recorderInfo( QWidget* object,
30                                                  const QString& propertyName,
                                                    const QPoint& pos )
    {
        RecorderInfo res;

35      if ( object->inherits("QTabBar") &&
             propertyName == "Name of Tab at Point" ) {
            QTabBar* bar = static_cast<QTabBar*>( object );
            QTab* tab = bar->selectTab( pos );
            res.insert( "id", QString::number( tab->identifier() ) );
40      }

        return res;
    }

45  QVariant MyPropertyExtension::value( QWidget* widget, const QString propertyName,
                                         const RecorderInfo& recorderInfo,
                                         bool& valid )
    {
        if ( propertyName == "Current Tab Name" ){
50          QTabBar* bar = static_cast<QTabBar*>( widget );
            if ( bar ){
                return QVariant( bar->tabAt(bar->currentTab())->text() );
            }
        }
55
        else if ( propertyName == "Name of Tab at Point" ) {
            QTabBar* bar = static_cast<QTabBar*>( widget );
            int id = recorderInfo["id"].toInt();
            QTab* tab = bar->tab(id);
60          return QVariant( tab->text() );
        }

        else if ( propertyName == "List of Top Level Windows" ) {
            valid = false; // Do not include information about widget being printed
65          QStringList names;
            QWidgetList* list = QApplication::topLevelWidgets();
            for ( QWidgetListIt it( *list ); *it; ++it ) {
                names << (*it)->name();
            }
70          delete list;
            names.sort();
            return QObject::tr("Top level names: ") + names.join( ", " );
        }

75      // We must return a null QVariant to indicate that this
        // property extension did not define the given property
        return QVariant();
    }

80  PropertyValueList MyPropertyExtension::value( QWidget* widget,
                                                  const QString& propertyName,
                                                  const RecorderInfo& )
    {
        PropertyValueList result;
85      if ( propertyName == "All Properties of Widget" ) {
```

```
            dumpProperties( result, widget );
        }
        else if ( propertyName == "All Properties of Dialog" ) {
            QObject* child = widget;
 90         while ( child->parent() && child->parent()->isWidgetType() ) {
                child = child->parent();
            }
            dumpAllProperties( result, static_cast<QWidget*>( child ) );
        }
 95     return result;
    }

    void MyPropertyExtension::dumpProperties( PropertyValueList& result,
                                              QWidget* obj )
100 {
        QMetaObject* meta = obj->metaObject();
        QStrList list = meta->propertyNames( true );
        for ( QStrListIterator it( list ); *it; ++it ) {
            result << PropertyValue( KDExecutorUtil::childName(obj), *it, obj->property(
105     }
    }

    void MyPropertyExtension::dumpAllProperties( PropertyValueList& result,
                                                 QWidget* widget )
110 {
        dumpProperties( result, widget );
        const QObjectList* list = widget->children();
        if ( list ) {
            for( QPtrListIterator<QObject> it( *list ); *it; ++it ) {
115             if ( (*it)->isWidgetType() ) {
                    QWidget* w = static_cast<QWidget*>( *it );
                    if ( w != widget ) {
                        dumpAllProperties( result, w );
                    }
120             }
            }
        }
    }

125
```

In Figure 8.2 the implementation file for the property extension is available. Lets look at each of the parts in details.

Line 14-19 tell KD Executor that there are properties called `Current Tab Name` and `Name of Tab at Point`, which should show up in the popup menu labeled `Example Extensions - No Plugin`. These property, however, only shows up if the widget in question inherits the class QTabBar.

Lines 20-25 tells KD Executor that it should offer properties called `All Properties of Widget`, `All Properties of Dialog`, and `List of Top Level Windows` for any widget. These will also be located in the popup menu labeled `Example Extensions - No Plugin`.

Lets for a short while jump to the lines 49-54. These lines prints out the property for `Current Tab Name`. The value returned on line 49 is the value the user will see in his output file or on standard output.

The property `Name of Tab at Point` is printed out in line 56-61. This property, however, needs to know which tab was under the point during recording, for that it uses

the `recorderInfo` argument to the method. This class brings over the id of the tab under the mouse during recording. Line 58 is the one reading the id out of the class.

The method `recorderInfo()`, which you can see on lines 29-40 is called by KD Executor at recording time, and must fill the `RecorderInfo` class we use in line 58 as described above. The method finds the tab under the mouse cursor in line 38, and inserts the id into the struct at line 39. This information goes into the XML file used during playback.

Returning to line 63-73, we see a property extension which really isn't a property in normal sense. This property extension namely prints out the names of all the top level widgets. The list is of course the same nevertheless which widget the mouse is clicked on, therefore we set `valid` to false in line 64, to tell KD Executor that it should not print information about which widget was selected.

Line 77 returns an empty QVariant, which tells KD Executor that this property extension did not contain a definition for the property in question. KD Executor will then continue querying other property extensions.

Lines 85-87, and 88-94 print out the property extension `All Properties of Widget` respectively `All Properties of Dialog`. They do so using the auxiliary functions `dumpProperties()`, and `dumpAllProperties()` which you can see on line 97-124.

As these two property extension prints out more than one property, we need to use the overloaded `value` function, which return a `PropertyValueList`. Items are inserted into the list in the method `dumpProperties` on line 104.

Putting the Property Extension to work

Next, you must activate you property extension. Doing so is pretty similar to activating an adapter. You can do it either by making it a dynamically loadable plugin, or by putting it into your source code, in the situation that KD Executor is compiled into your application.

To do it in your source code, simply create an instance of your plugin, and call `KDExecutorFeatures::addPropertyExtension()` with your instance as an argument.

To create a plugin, compile your property extension as a library, with the preprocessor define `QT_PLUGIN` set. (This can be done by adding a line saying `CONFIG+=plugin` to your QMake project file You must also add a define for `KDEXECUTOR_DLL`, which in QMake syntax would be the line saying `DEFINES += KDEXECUTOR_DLL`.. This library needs to be put in `$KDEXECUTORDIR/lib/plugin`.

Examples of a property extension compiled as a plugin is available in `examples/properties-plugin/property-extension`, and an example of one compile into the application is available in `examples/properties-no-plugin`.

Chapter 9. Frequently Asked Questions

9.1.

KD Executor Fails to Record Print Dialogs?

On Windows and MacOS X, the print dialog is a system dialog. This has the consequence that no Qt-level events go to this dialog. Since KD Executor is recording events at the Qt level, it is simply impossible for KD Executor to record anything for the print dialog.

9.2.

KD Executor Fails to Record Drag and Drop?

Unfortunately it is not possible for KD Executor to record drag and drop.

9.3.

KD Executor Fails to Record XXX?

As described in the chapter on Adapters, a number of adapters are in place during recording, for recording the intention rather than the action. These adapters might fail to record the correct intention in certain situations. To test if this is the case, disable all adapters in the configuration dialog. If that solves your problem, try enabling individual adapters till the problem shows again, and you have found the culprit.

9.4.

My Program Works Differently when KD Executor is Enabled?

This might be due to an adapter, see the previous FAQ item.

9.5.

Will KD Executor stay binary/source compatible like Qt does?

Binary compatibility is an important issue if you are developing a library like Qt. If a library upgrade is not binary compatible, all applications using it will need to be recompiled with the new version.

Binary compatibility, however, comes with a large price tag. To stay binary compatible you have to follow a number of rules that make it hard to add new features, to fix certain interface bugs, etc.

All current and potential customers have expressed intentions of using KD Executor in no more than a few applications. For this reason we feel that our cus-

tomers will get a better product in future upgrades when we do not limit KD Executor to stay binary compatible.

Upgrade versions of KD Executor will be source compatible with previous releases unless a problem can only be fixed by breaking source compatibility. Should this happen, we will ship thorough instructions on how to update your source code.

9.6.

Will the script language change in the future?

At Klarälvdalens Datakonsult AB we realize the importance of future versions of KD Executor being capable of reading the scripts that older versions have written. We will therefore do our very best to ensure that this will always be the case.

If we for some reason need to update the script language in an incompatible way, we will—if possible—(1) ship a program that can convert all your script files in one go, and (2) build into KD Executor the capability to convert the files on the run.

One thing we can, unfortunately not promise, is that Trolltech does not change the underlying event system in an incompatible way. This happened between Qt version 3.0 and 3.1 for keyboard accelerators. In such a case, we will try to take the steps as described above, and, if this is not possible, we will at least describe the situation thoroughly for our users.

9.7.

Are there any KD Executor-related mailing lists?

An *announce mailing list* where Klarälvdalens Datakonsult AB will inform subscribers about new versions can be found at http://klaralvdalens-datakonsult.se/mailman/listinfo/kdexecutor-announce.

A *general KD Executor discussion list* can be found at: http://klaralvdalens-datakonsult.se/mailman/listinfo/kdexecutor-interest.

9.8.

Can I use KD Executor with tools like Rational Purify, Rational PureCoverage, or valgrind?

Sure, a KD Executor enabled application is nothing but an application seen from the above mentioned tools. For example, you can use PureCoverage to check your KD Executor scripts for completeness, or combine KD Executor and Purify for a more thorough regression testing.

Chapter 10. The Format Of The Script Files

The following describes the content of script files that KD Executor generates.

The topmost tag is called `Events`, except in the situation where KD Executor is working with a socket—in this mode, the topmost tag is not present.

Below the topmost tag each element represent an event. The elements are described in the following sections.

Each element in the script file has two mandatory attributes:

name	This is the name of the widget to which the event is targetted. Names are constructed from the parent/child path to the widget, for each step the `QObject::name()` name is used, when necessary postfixing it with a number if several objects with the same name exist in the given path of the given class.
sleep	The period to sleep before this event should happen.

Low-level Events

MousePress, MouseRelease, MouseMove, MouseDblClick	These four items represent mouse press, mouse release, mouse move and mouse double click events.	
	x (optional)	the x coordinate of the mouse.
	y (optional)	the y coordinate of the mouse.
	button (optional)	the button which has triggered the event (See QMouseEvent::button for details). Allowed values are `No`, `Left`, `Mid`, `Right`
	state (optional)	the state just before the mouse event occurred (see QMouseEvent::state for details). The value is a comma separated list of the follow-

		ing items: `Left`, `Mid`, `Right`, `Shift`, `Control`, `Alt`
KeyPress, KeyRelease, Accel	These three items represent key events. KeyPress is recorded for keys being pressed down or held down (for repeated key events), KeyRelease for the key being released, and finally Accel for accelerator keys-the user should rarely need to create such an event. See QKeyEvent for more details on the parameters of this item.	
	key (optional)	This is the key in question specified as the argument to the constructor of the class QKeySequence. Examples include `a`, `Down`, and `F1`.
	intkey (internal)	KD Executor will try to print the key in ascii syntax, but if it fails it will insert the key code as an integer value. Users writing script files by hand should refrain from using this attribute.
	ascii (optional)	ASCII code of the key.
	state (optional)	The state of the shift, control and alt key before the key event occurred. Example: `Shift, Control`.
	text (optional)	The Unicode text that the key has generated.
	repeat (optional)	a boolean value describing whether this is a repeated event (can only occur with key presses).
	count (optional)	The number of single keys.
ResizeEvent	A resize event. If you call KDExecutorFeatures::setResizeTopLevels(false) (the default), then KD Executor will not send this event to top-level windows.	

	width	the new width of the widget
	height	the new height of the widget

MoveEvent

Tells the widget that it has been moved to a new position. If `KDExecutorFeatures::setMoveTopLevels(false)` is called (the default), this event will not be sent to top-level widgets.

x	new x coordinate
y	new y coordinate

WheelEvent

Represents a wheel event.

x (optional)	the x coordinate of the mouse.
y (optional)	the y coordinate of the mouse.
delta	the number of steps the wheel has moved.
state (optional)	the button state of the shift, control, and alt buttons. Example: "Shift, Control".
orient	the orientation of the wheel event. Either `horizontal` or `vertical`.

FocusEvent

Represents a focus- in or focus-out event,

type (optional)	the type of the focus event. Either `FocusIn` or `FocusOut`. The default is `FocusIn`
reason (optional)	the reason for the focus event. Available options are: `Mouse`, `Tab`, `Backtab`, `ActiveWindow`, `Popup`, `Shortcut`, `Other`. The default is `Other`.

WindowActivate, WindowDeactivate	These two events take no arguments.	
ContextMenu	Requests a context menu to be shown.	
	x (optional)	the x coordinate of the mouse
	y (optional)	the y coordinate of the mouse
	state (optional)	the state just before mouse event occurred (see QMouseEvent::state for details). The value is a comma-separated list of the following items: `Left`, `Mid`, `Right`, `Shift`, `Control`, `Alt`,
	reason (optional)	the reason for this event. Valid values are `mouse`, `keyboard`, and `other` (the default).

High-level events

The adapters do of course also generate XML. The syntax for this is unfortunately not currently documented. It should, however, be fairly easy to read from the adapters them self.

If this information is needed, please contact kdexecutor-support@klaralvdalens-datakonsult.se, and we will extract the information needed.

KD Executor Events

Version	The very first element in the script file must be a Version element specifying the version number of the KD Executor script language.	
	id	the version number.

ScreenDump	Creates a screen dump of the specified widget.	
	fileName	the name of the file to save the screen dump to
Include	Executes another script file in place of this file. The other script file needs to have an Events tag at the top.	
	file	the name of the file containing the script.
PrintProperty	Prints a property for a widget (see the chapter Chapter 4, *Printing Interface Properties* for details). This XML tag may have a text in it. The text is printed out when printing the property.	
	x	the x coordinate within the widget
	y	the y coordinate within the widget
	property	the name of property to print
Sleep	Asks the playback unit to sleep for an absolute period of time-i.e., even if timing is compressed. See KDExecutorFeatures::setCompressTime and KDExecutorFeatures::Sleep.	
	msec	the number of milliseconds to sleep.