v0.8

User's manual

1 Table of contents

Table of Contents

2 Introduction

2.1 What jDiffChaser is

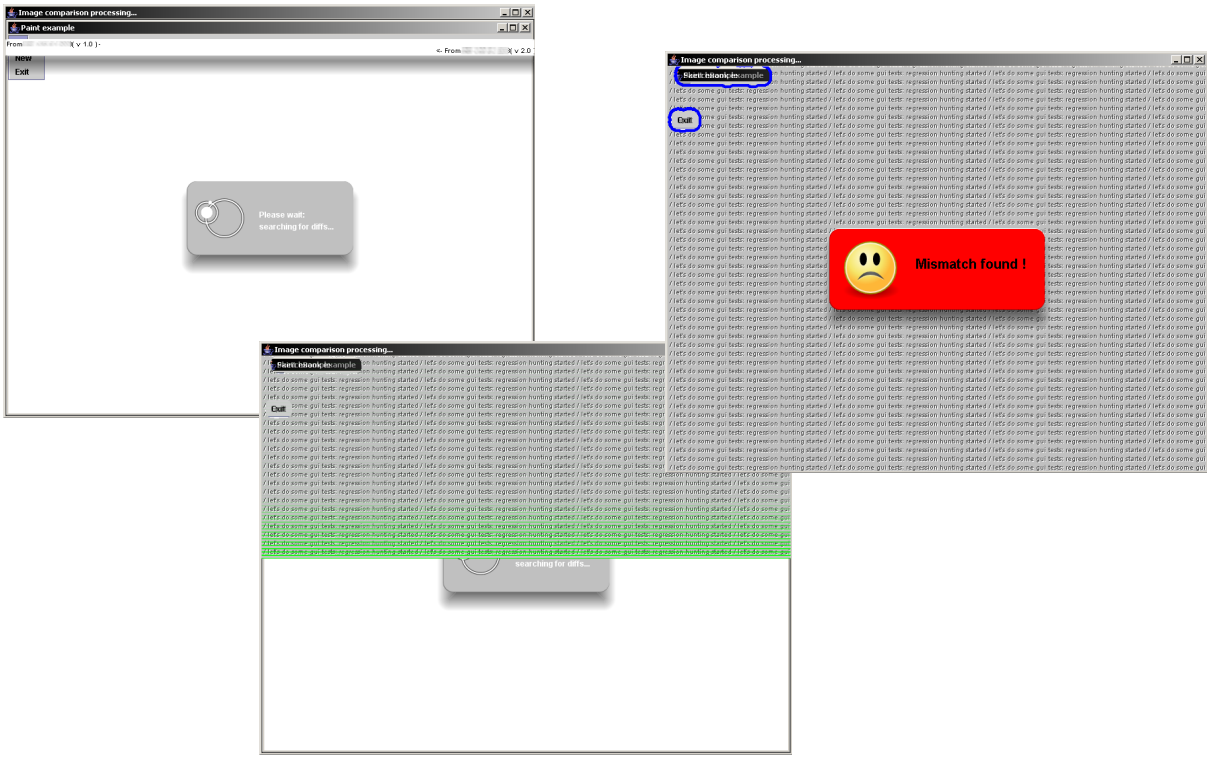
jDiffChaser is a GUI comparison tool that automates difference detection between same screens of different versions. You can easily record scenarios (optionally define zones of the screens to ignore during comparisons) and play suites of them on two different versions of the same Java Swing application: differences are then listed in a web page report.

You can execute scenarios using two modes:

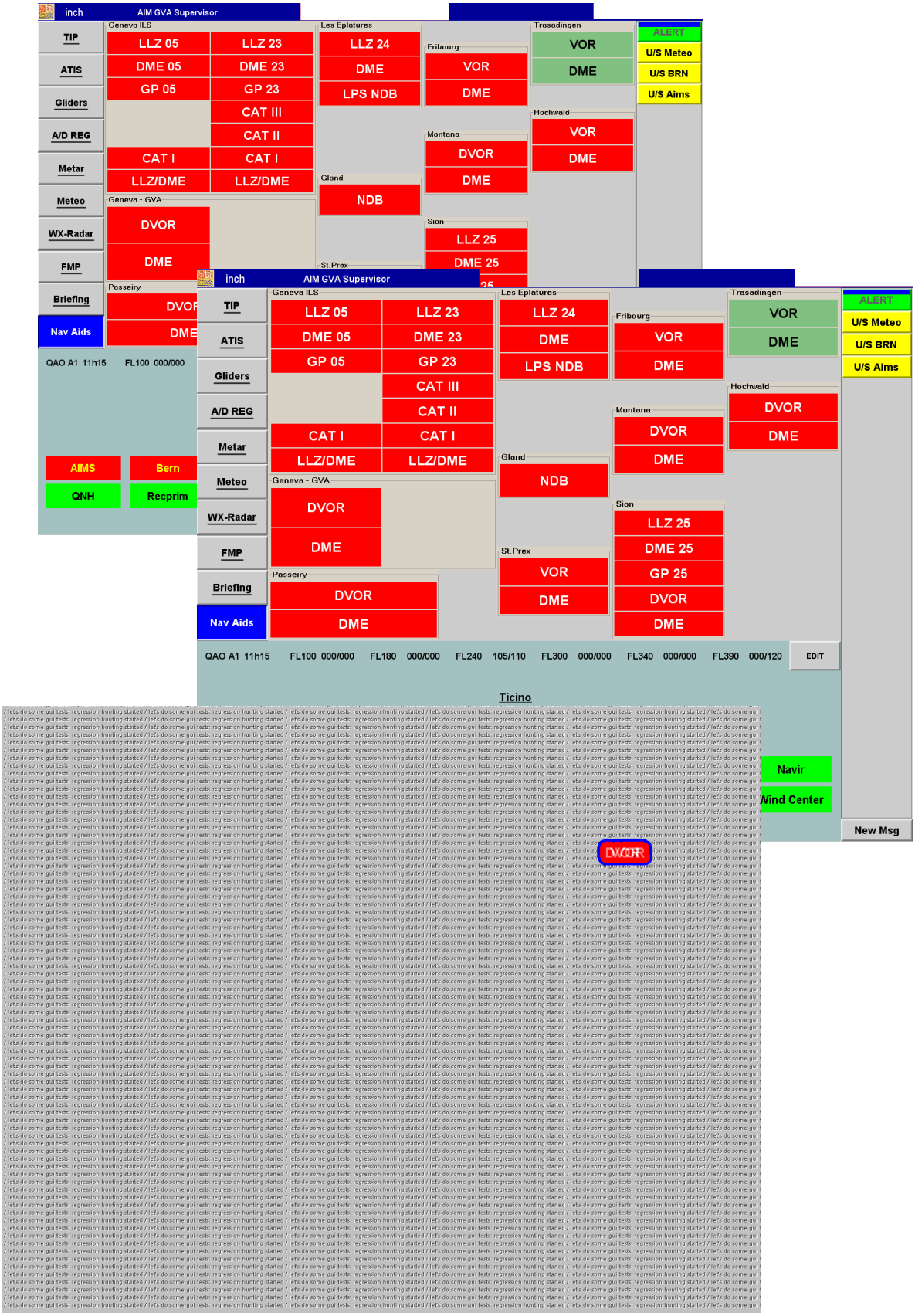
- sequential mode: you have only one test-host, each scenario is executed first upon the old version of your software then it is executed upon your latest software version.
- parallel mode: you have two test-hosts, each scenario is executed upon both versions (old and latest) of your software at the same time. This is useful when your application uses real time data.

2.2 Screenshots

A simple running comparison session example (the **Quick Start** one):



A more complicated result (taken from a collaborative air traffic control application):



2.3 Sounds familiar?

Let's have a look at two scenarios which jDiffChaser could have helped developers to avoid:

Scenario 1:

User: “Hi dev team, just a question: why do many of the buttons labels of the new release do not display entirely, it has some points at the end of the label?”

Dev team: “Strange, we did some font size changes but have tested all pages of the application in order to visually check that buttons sizes were correctly adapted. At least we thought to have... You say you have this problem on all pages? It seems to be impossible thanks to our tests...”

User: “Actually no, only on the hidden *Advance configuration* page you can access only in expert mode”

Dev team: “Damned, you're right. We probably forgot to look at this page, that's our fault ... There is definitively too many panels in this application to test all of them before each release :(.”

Scenario 2:

User: “Hi dev team, it seems the green *Validate* button of the new version validation page has a darker background than in previous versions. It makes the text less visible on it”

Dev team: “Ha ha... no way: no development has been done concerning this button for latest release”

User: “Humm... perhaps. But have a look with me, don't you see it is difficult to read the label?”

[One day later...]

Dev team: “Hey! User!... You were right... actually there was a tricky contextual bug in our button states that made the validate button have the *disabled* mode color even if its was enabled. Sorry about that.”

2.4 What jDiffChaser is not

jDiffChaser is not a testing tool as it is not able to decide what is correct or not. There's no assertion handling in it. This software can “just” help you finding some regression points your eye may have not seen. But in any case you can't only rely on its results to be sure there's nothing bad in your new release GUI. You still need to check the results every day, but one thing is sure: jDiffChaser reports differences with such a highlighting that it speeds a lot this tedious task of visual checking.

2.5 Why this tool?

2.5.1 As a regression support for GUI testing

Many of really good GUI testing tool already exist but many of them can't “see” the application rendering. For example you can test that a button text is “Confirm” but you can't test that it is entirely visible. Have you ever had this “Confi...” label (with the “...” at the end of it) when your JButton is too small to embed this text value? Have you ever seen the JScrollPane Horizontal scrollbar appearing because the contained panel was bigger than expected although you though not?

jDiffChaser also can't detect that something is correctly displayed within a new application, but as long as you have validated a version with end-users, it can check that the new version is visually correct, comparing screens between both releases using the validated one as the reference. That's why jDiffChaser's preferred action field is regression detection when traditional GUI testing tools are used to test the new features.

2.5.2 A “hacked” usage: as a feedback tool

Some development teammates sometimes have some difficulties communicating together. Thus some new features implemented by some of the developers sometimes remain unknown by others. It appeared that jDiffChaser has filled this communication gap into some teams, highlighting the new features. Some developers checking the report that was published onto a web server of the company intranet became aware of those changes. I know that this shouldn't occur in a development team, but that's a fact, this situation exists in many companies. jDiffChaser wasn't initially created for this task, but anyway, if it helps...

2.6 The requirements

jDiffChaser can only execute rich clients Java (TM) Swing applications that you can modify code. That's because you will need to add some little pieces of code dedicated to jDiffChaser in the application you want to compare screens. That's only about 4 lines, nothing more, but they are necessary until we will find a better solution in one of the next jDiffChaser versions. You will need at least the 1.4.2 Java (TM) version (and +) using Windows operating system and 1.5+ with other platforms such as OS X and Linux.

If you want to compile jDiffChaser and run its sample tests, you will also need Apache Ant.

2.7 History

jDiffChaser has been created by our development team working on a collaborative environmental data software for an air traffic control company. The software (in production since 2005) displays a lot of information grouped on many panels viewed by controllers who can have many roles. Such a software emphasizes the importance of chosen fonts, colors and components visual behaviors because each of those graphical information has a business meaning and can be contextual, depending on roles. That's why it is as much important to test the content as to test the visual aspect of what is displayed, in order to avoid what we used to call “visual regression”. We, responsible for this application, decided to create a tool

to help doing this checking because we didn't find any existing package to do so as traditional GUI testing tool were not dedicated to such a task.

It was clearly established from the beginning of its development, that the purpose of this tool was a supplementary way to find unwanted differences between versions in addition of many tests processes that were already used, mostly traditional human ones. Our team never wanted to exclusively rely on this tool, “real eye” tests still had to be done for critical reasons. The keywords were “early detection”: the goal was to detect and fix quickly any regression. The tool had to be easy to use to record new scenarios and had to be able to be automated in order to play many screens tests.

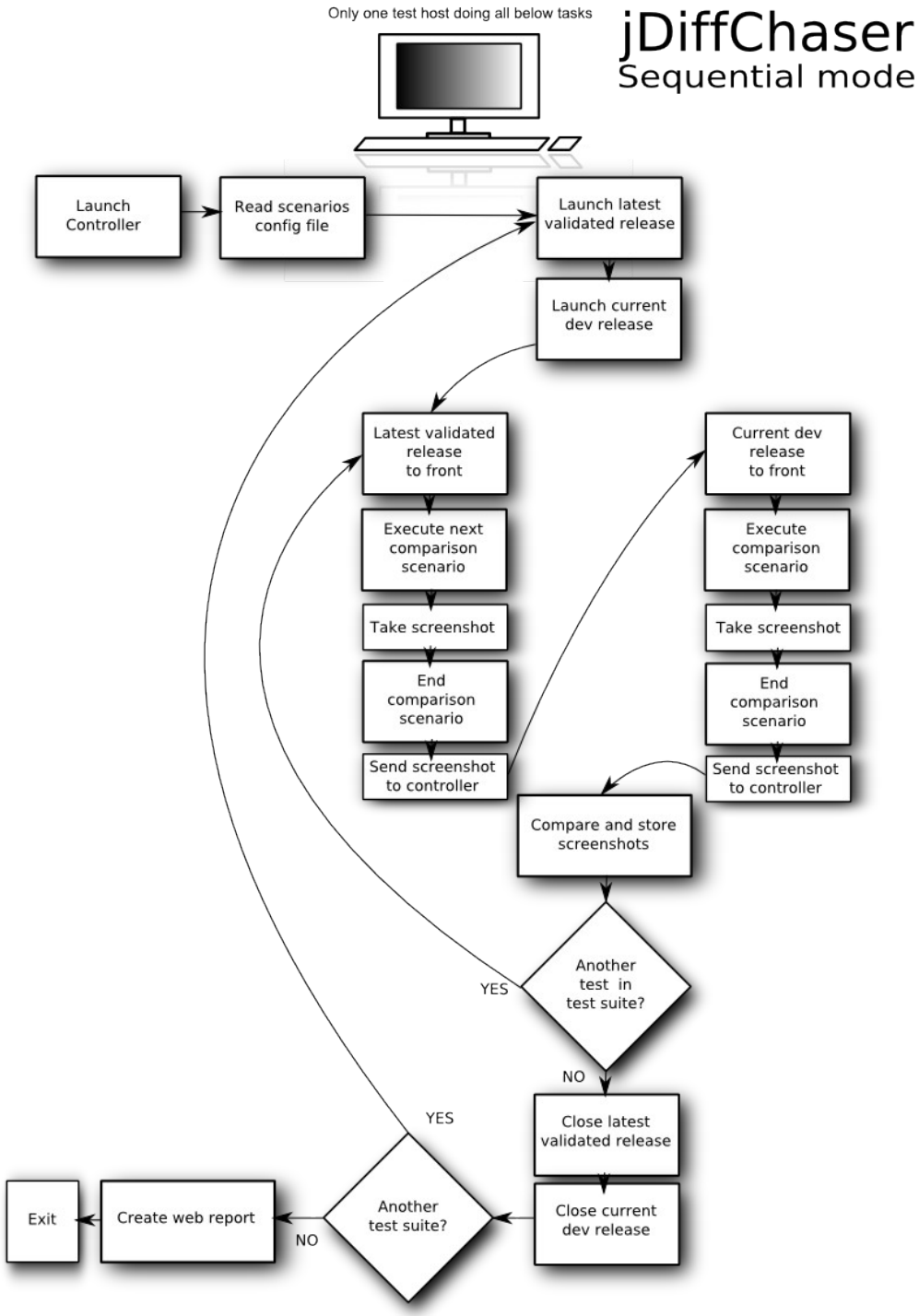
Our team now uses this tool since spring 2006. Every new release of our environmental data software becomes the new jDiffChaser reference after the traditional human (customer) validation occurred. Then, every night, our jDiffChaser scenarios are played upon the current development version to verify that no regression has emerged.

During winter 2006/07, we decided to create a more “generic” packaged version of our tool in order to publish it as an Open Source Software.

3 Sequential or parallel playing mode?

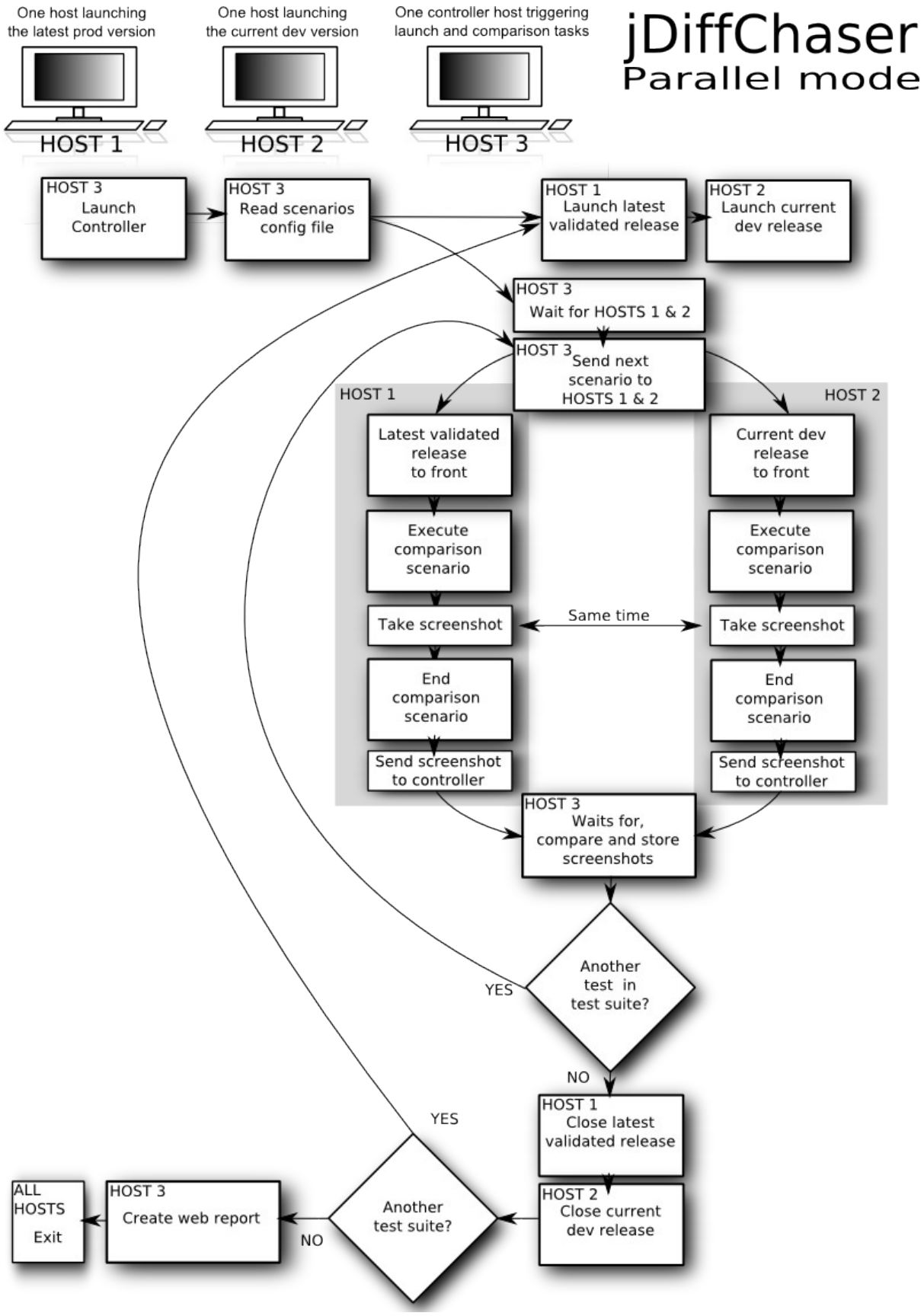
The way you will use jDiffChaser depends on how your application works. Let's suppose you have a standalone application that doesn't depend on peer's application nor real time data: you will work in a sequential playing mode. Let's detail how your jDiffChaser system will operate.

3.1 Sequential playing mode



Now that we have seen how jDiffChaser operates with a “classic” standalone application, let us detail how jDiffChaser compares locally two screenshots taken from two distant computers screens at a given time: that was the case we had to deal with our collaborative environmental data software.

3.2 Parallel playing mode



4 Quick start

4.1 The sketchbook sample

When you download the jDiffChaser package, we provide you the sketchbook sample. Thus you can play with a simple application and discover how a scenarios playing is done and how you can record some new scenarios. There are two virtual versions of this application:

- org.jdiffchaser.samples.sketchbook.**version1**.SketchBSample has the role of validated release, it will be our reference for visual tests
- org.jdiffchaser.samples.sketchbook.**version2**.SketchBSample has the role of what could be the current development version

Of course in real life you would have probably two different set of jar files corresponding to both versions. We admit our quick start example is very simple ;)

The distribution package includes the following ant scripts:

- ***`run-recorder-sketchbook-sample`*** : starts a recording session using the **version1** of *SketchBSample.*
- ***`run-localplayer-sketchbook-sample`*** : let you choose a scenario and play it locally (to test the scenario events) using the **version1** of *SketchBSample.*
- ***`run-guitests-sketchbook-sample`*** : plays the whole set of test-suites defined for *SketchBSample*. As windows handling and rendering depends on operating systems, this script choose and runs scenarios that were recorded running the *SketchBSample* on either a Microsoft Windows or MacOS X system.

When writing this documentation, no *SketchBSample* scripts have been recorded at all on a Linux system, but have a look inside the distribution package, today they may exist.

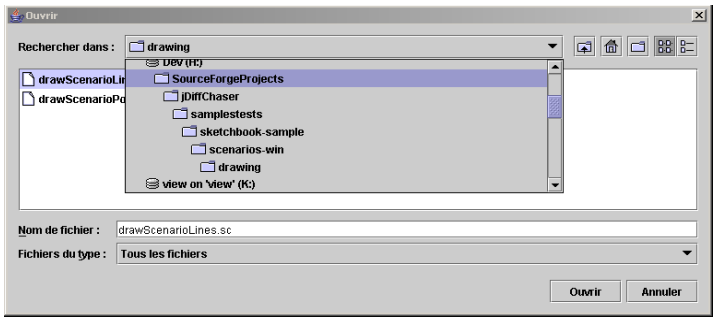
4.1.1 Play the distribution sample scenarios

In this section we will have a look at how jDiffChaser....chases.

In a console, when at the root of the jDiffChaser, type:

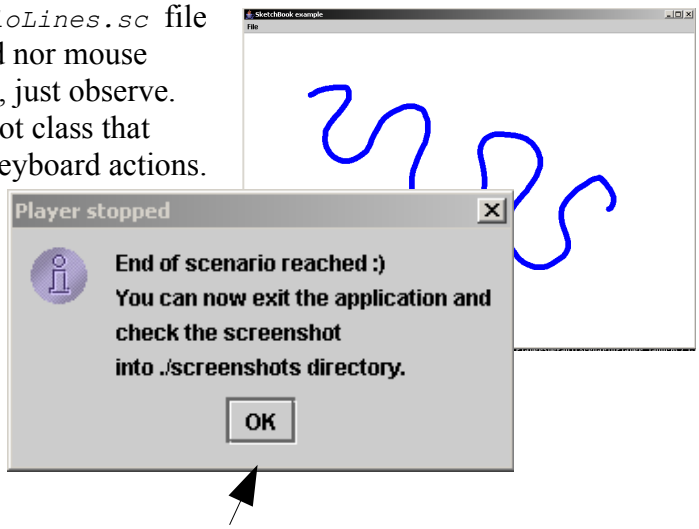
`ant` (just in case you've not already build the project) then,

```
ant run-localplayer-sketchbook-sample
```



This previous dialog appears, asking you to choose the scenario to execute and validate. No comparison will be done during this execution, this is just a step you can do to validate the actions your scenario will trigger.

Let's choose the `drawScenarioLines.sc` file and don't touch your keyboard nor mouse anymore during the execution, just observe. jDiffChaser uses the java Robot class that commands your pointer and keyboard actions. Unfortunately you still can use your mouse and keyboard when a Robot is executing. This can result in a badly altered scenario. So once a scenario is running on a computer, don't touch your keyboard nor your mouse anymore ;)



Once the scenario is finished, this dialog box tells you can check the resulting scenario screenshot in a given directory. The screenshot represents what will be compared during a comparison session.

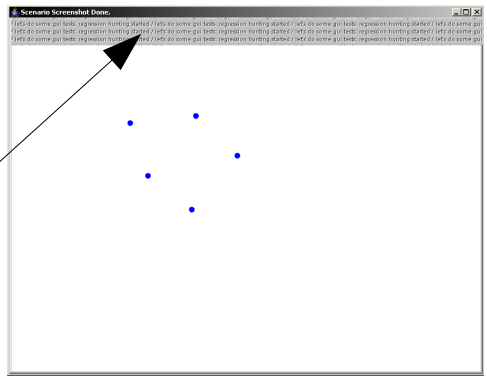
Let's say we are ok with how this scenario triggers actions. Now we are going to execute the whole set of test-suite the jDiffChaser team has prepared for you.

Run the following command and once again, just observe, this can take some time as it uses real actions time and delays:

```
ant run-guitests-sketchbook-sample
```

Once the whole comparison process is finished, it's time to detail what you have seen.

First, note that the tasks flow followed by jDiffChaser in this example is the one described in the **3.1** section of this documentation, the sequential one. First, you have probably noticed that the scenarios aimed at comparing the menu content, the action of the ***New*** menu entry, the drawing of lines and finally the drawing of single points. One thing that you may have observed during screenshots is that some parts of the screenshots were filled with a strange gray pattern as the following image illustrates.



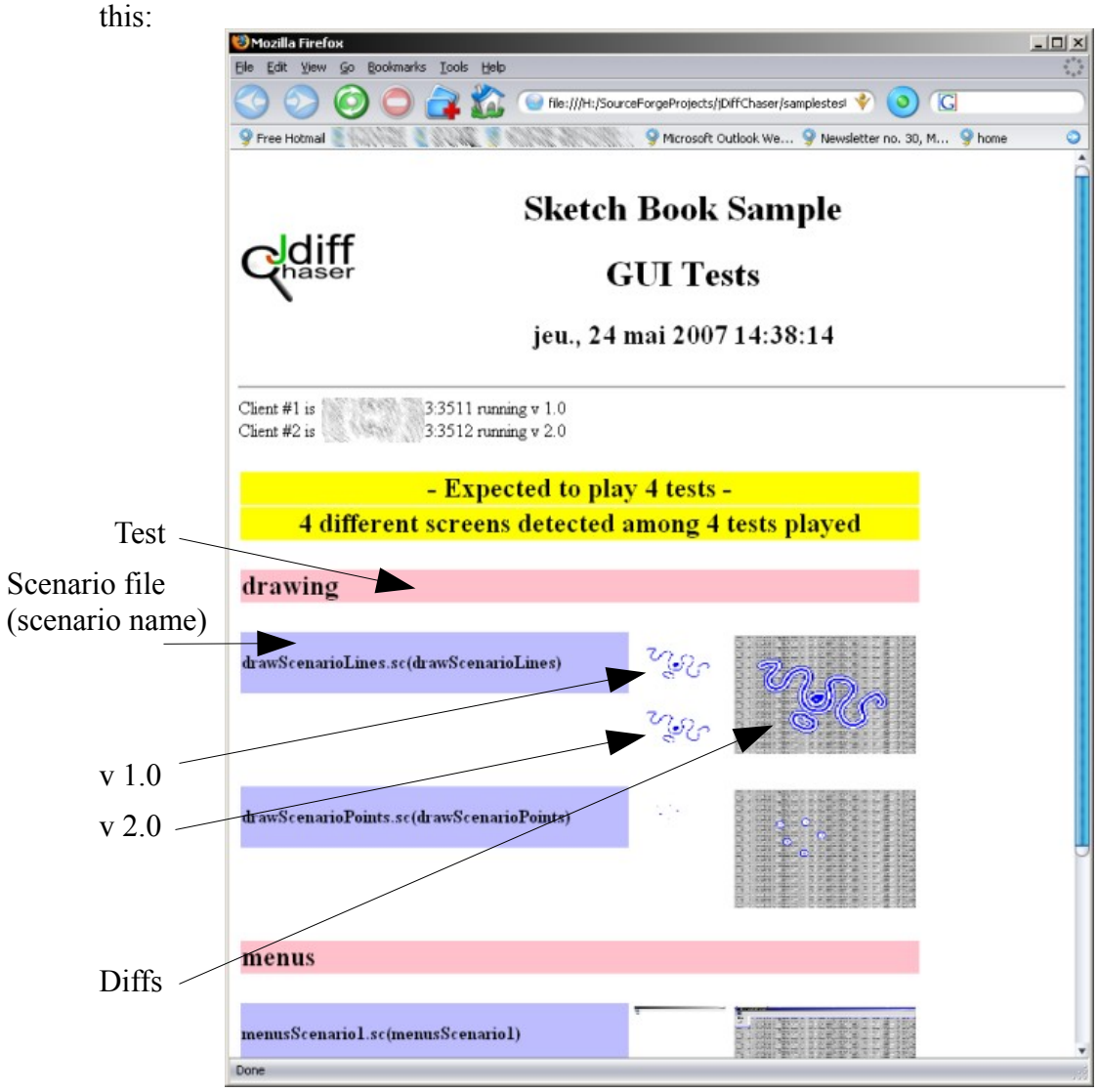
Zone that will be ignored during comparison

Actually, during some comparisons, we want to focus on some part of the screen, no matter what happened to other parts of it. That's why it is possible, when recording a scenario, to ignore some zones during a comparison. We will see later how to do this, but let me give an illustration of that requirement.

For example, concerning the screenshot you saw on the previous page, it's a screenshot done during the scenario that tests the “single point drawing” feature. We didn't want to verify if the window title was ok nor if the file menu label was right... We just wanted to test that the result of our click operations was similar on both drawing panels, old and new ones. So we ask to just compare that part of the screen.

4.1.2 The results: SketchBSample have regressed !

Now that you've executed all desired comparison tests on both versions of SketchBSample, it's time to have a look at the results. When all tests suite have been executed, the jDiffChaser controller creates a simple html report. The SketchBSample can be found in the `jDiffChaser\samplestests\sketchbook-sample\failed` directory. Open the `index.html` file and you will see something like this:



We have just concluded that the SketchBSample application had regressed: that's true if we consider that the single point drawing feature seems to have been broken. But if we look more precisely at the differences found by jDiffChaser during the `drawScenarioLines.sc` scenario, we realize that the new version draws the lines with anti-aliasing whereas the previous doesn't. This is clearly not a regression but an improvement. The presence of such a result typically illustrates what we called in the **2.5.2** section “the feedback tool”: we keep a visual trace of what have changed or evolved.

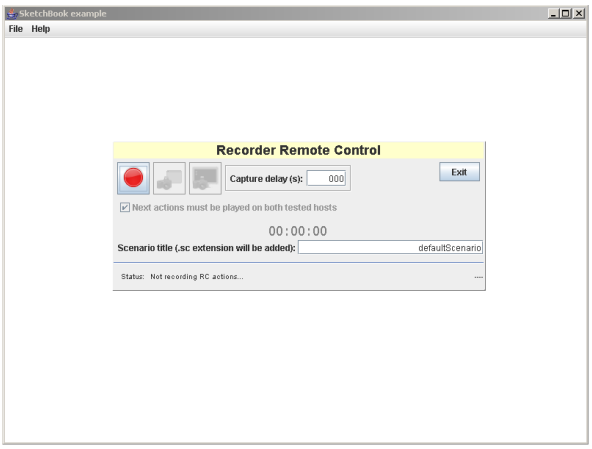
Some will then decide to remove this scenario from the list (in order to remove the difference from the result) whereas others will decide to keep this scenario to have a way to present the differences between versions when requested (by the customer for example) or to make the whole team know the improvements between two versions.

4.1.3 Record and add a comparison scenario

Now that you know how jDiffChaser plays scenarios, you probably want to know how to record an additional scenario and add it to an existing test. Let's suppose you want to add a comparison of screens including both lines and points (ok, you're right, that's not very complicated but the *SketchBSample* application is very basic, so...). Note that if you're using Os X, see the **7.2** section before doing this tutorial.

Run the following ant task:

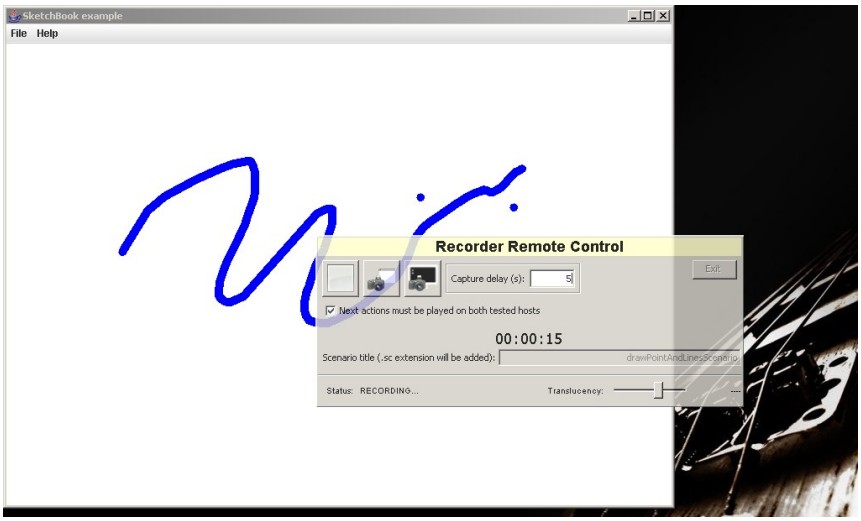
```
ant run-recorder-sketchbook-sample
```



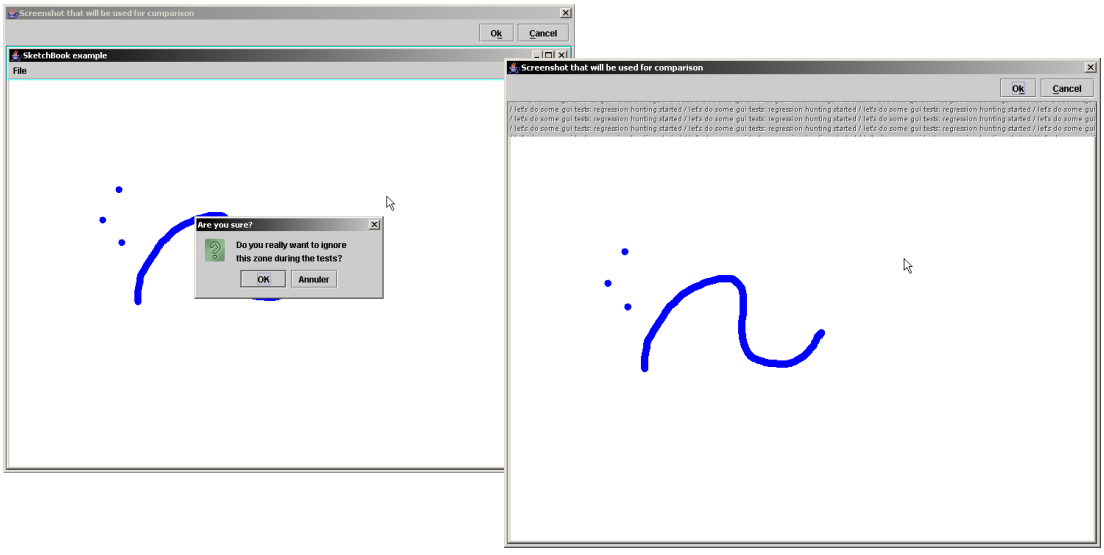
This results in displaying a *SketchBSample* frame as well as the scenario recorder frame we will call the remote control. The remote control is an “always on top” frame totally self-supported (have its own JVM) for reasons I will explain later, not in the Quick Start section. Just have in mind that the remote control won't interfere with how your application is working.

First, type a scenario file name in the dedicated textfield. Let's choose the following name: *drawScenarioPointsAndLines* (this will create, at the end of the recording, a *drawScenarioPointsAndLines**.sc*** file). Once you feel ready to record, press the record button (the red one) and start to draw points and lines over the drawing panel. At any time, you can move the remote control during the scenario

recording by dragging its frame (start dragging the frame on any part of it), any action within the remote control is, of course, not recorded. You may have noticed that the timer had started when you had pushed the recording button. You probably have guessed this gives you the elapsed time since the scenario has started.



Once you have finished drawing points, curves and lines, push the application capture button (with a camera icon over an application window). The second camera button (with a screen behind the camera) is for fullscreen screenshots, we won't use it now but keep in mind that it is useful for multi-windowed applications and applications using the MacOS finder menu bar. The remote control hides itself after the delay given in the dedicated textfield (right side of capture buttons, this allows to record pop-up menu display on OS such as MacOS: launch the delayed capture and do the action), then the screenshot dialog appears, waiting for your actions... Why actions? This is the step where you can decide to ignore zones to compare. In our example, we will hide the whole title and menus area as we just want to validate that points and lines are correctly rendered when drawn together on the drawing panel. To do this, click at the upper left corner of the screenshot and drag your mouse until the upper right corner of the drawing area. A dialog box asking you to confirm the location of a “Ignored Zone Area” appears: click Ok when you think your area is correct.



Then your area is filled with the “Ignored Pattern”. Of course, you could add more areas to ignore but we will stop here for our example. You can now click on the OK button of the screenshot dialog, this closes the dialog. If you were not sure of doing a screenshot at this time of the scenario, you would have clicked on cancel. The scenario would have continued and you could have triggered the screenshot later in it. Anyway, let's go back to our example.

After having closed the screenshot dialog, the remote control timer continues (it had been stopped during the screenshot settings step) and it's time to clean our area, just in case another drawing test is done after this one. Go to the *SketchBSample* menu bar, click on the file menu and click on new. This results in erasing the drawing area content. Now you can stop the scenario recording by pressing the stop button (a gray square button that appeared at the record button location when the recording started), the timer also stops. Recording is done.

Now you can find the resulting file in the directory *jDiffChaser\scenarios\sketchbook-sample* (directory specified in the ant target , we'll see later how to write such a target).

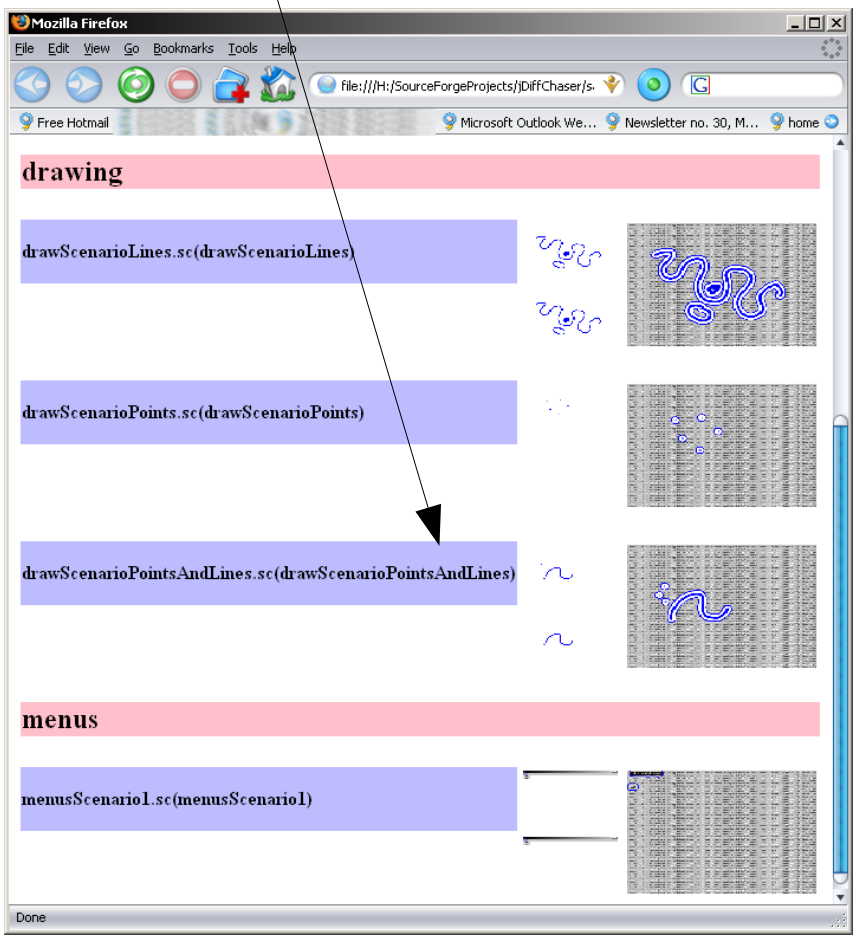
Move this file into the

jDiffChaser\samplestests\sketchbook-sample\scenarios-win\drawing (adapt if osx)

directory and launch the test-suites ant task another time:

```
ant run-guitests-sketchbook-sample
```

Once the test-suites have been executed, you'll see at the end of the report the new difference highlighted (containing points and anti-aliasing differences)



4.1.4 The configuration file

If you look at the *tests-sketchbook-sample-win.xml* configuration file in the *jDiffChaser\samplestests\sketchbook-sample* you will see:

```
<?xml version="1.0" encoding="UTF-8"?>
<test-configuration>
	<report-title>Sketch Book Sample</report-title>
	<first-host>
		<ip>localhost</ip>
		<port>3511</port>
	</first-host>
	<second-host>
		<ip>localhost</ip>
		<port>3512</port>
	</second-host>
	<scenarii-base-directory>samplestests/sketchbook-sample/scenarios-
win/</scenarii-base-directory>
	<failed-base-directory>samplestests/sketchbook-sample/failed/</failed-
base-directory>

	<test-suite>
		<gui-test>
			<scenarii-directory>menus</scenarii-directory>
		</gui-test>
		<gui-test>
			<scenarii-directory>drawing</scenarii-directory>
		</gui-test>
	</test-suite>

</test-configuration>
```

Here are the tags descriptions:

- <report-title> : as you guess, it's the title appearing at the top of report
- <first-host> and <second-host> : describe the two hosts that run both instances of the application. In our example, it is the same host. We just cared about choosing different ports because we were running all versions on the same host
- <scenarii-base-directory> : this is the root directory of all tests. Each test will be a subdirectory of this one
- <failed-base-directory> : this is the root directory of all failed comparisons. Each failed comparison will be a subdirectory of this one. This directory also contains the report file
- <test-suite> : contains all tests to play among the test directories found under the <scenarii-base-directory>. The application is restarted after each test-suite, except for the last one
- <gui-test> : represent a test; a set of comparisons found in a subdirectory of the <scenarii-base-directory>
- <scenarii-directory> : this is the name of the <scenarii-base-directory> subdirectory containing scenarios to play. Scenarios are played according to the Operating System sorting rules (most of the time alphabetically)

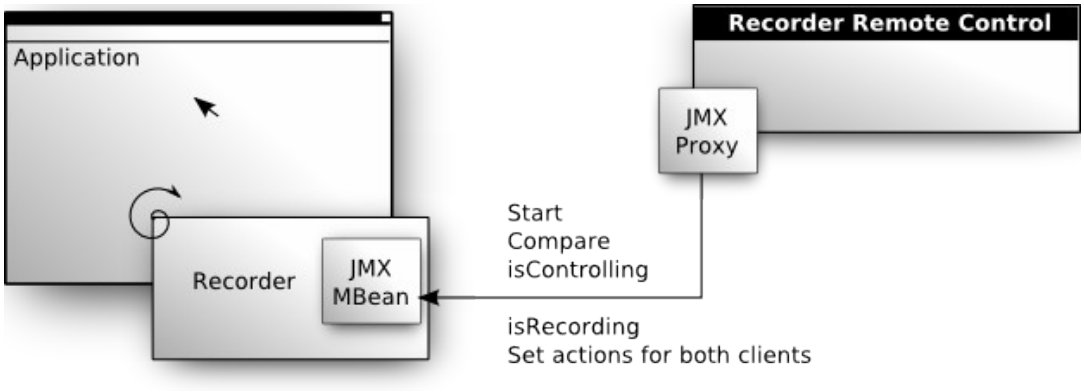
Those tags and options are not the only ones that exist in a jDiffChaser configuration file. Those are the basic ones and others are detailed later, not in the Quick Start section.

I invite you to observe the source code of the ant tasks you've used in this section in order to understand how to launch jDiffChaser tasks. Now I'm going to explain jDiffChaser principles in order to better understand the system.

5 How it works

5.1 How scenario recording works

As you will see during the next parts of this documentation, jDiffChaser uses a lot of Java JMX Remote API features for various aspects of the system. One of those aspect is the way the Remote Control frame controls the application to record events from. Basically, here is how the recorder operates:



But why using JMX and having a true standalone Remote Control frame? That solution was chosen in order to be able to capture screens with modal dialogs. This remote control has not always been using JMX: in the early versions of jDiffChaser, this frame was running within the same JVM as the recorder, thus the application. All was working well until the day we had to deal with scenarios including modal dialogs in them. So we had to find a way to less interfere with the tested interface; JMX Remote API in the Remote Control was a possible solution.

Consequently, when launching a recording session, you have to launch two Java processes, one for the Application[+Recorder] and one for the Remote Control. This is easily done using, for example, Apache Ant software. Have a look at the jDiffChaser *build.xml* file, the `run-recorder-sketchbook-sample` task is the concerned one.

Concerning the Application[+Recorder] couple, the *org.jdiffchaser.testing.DefaultRecorder* class takes three application arguments:

- the main class of application to test (for the sample it was: *org.jdiffchaser.samples.sketchbook.version1.SketchBSample*)
- the directory where to store taken screenshot
- the JMX port the Remote Control will communicate with

If you need to pass specific arguments to your application, you have to create your own Recorder. See the *DefaultRecorder* class code to see how to implement it, this is not a difficult to do so.

You also have to pass one JVM arguments:

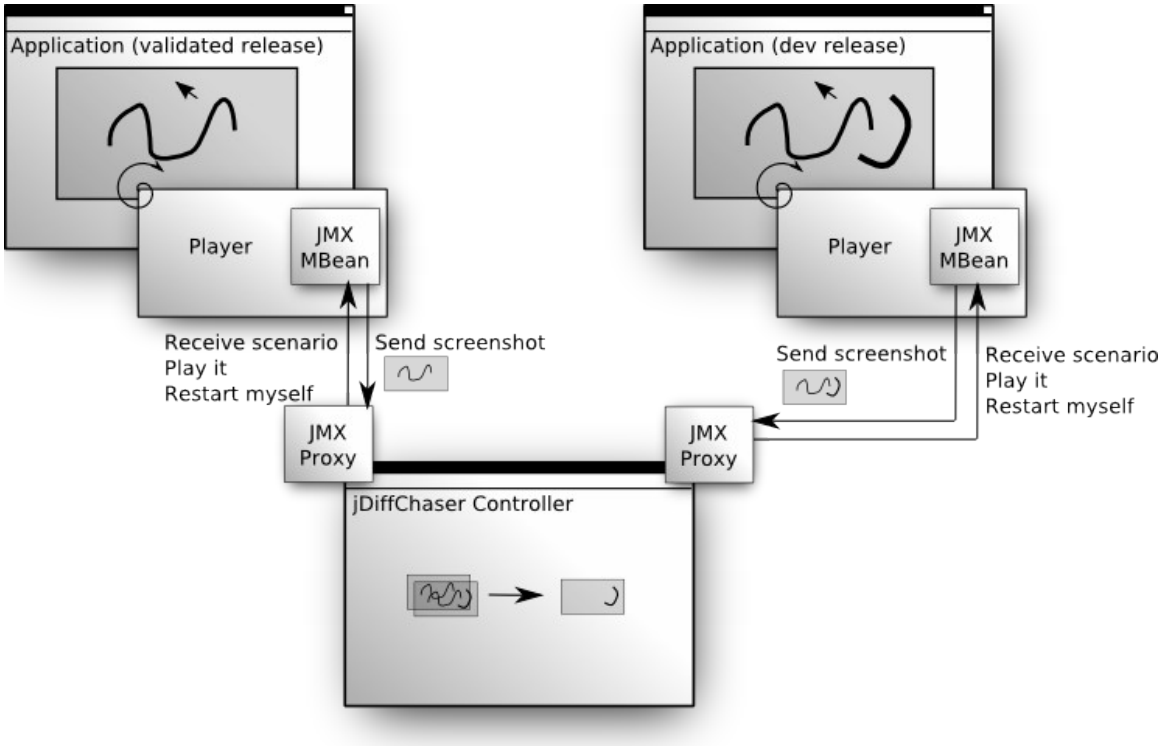
- *-Djava.library.path* indicates where to find some native library needed to make the Remote Control frame always on top for JVM before version 1.5.

5.2 How screen capture works

Using Java Robot class. I know you are wondering "why using Robot to take screenshots when it's so popular how to capture a frame rendering of a Java standalone application: just render it in a buffered image?". You are right, for most cases, it is sufficient. But how can you capture the screen state when you have many dialogs rendered (I'm not speaking about internal frames, but real independent dialogs)? The better way to do this using Java is the Robot class. Keep in mind we need to capture what we see, not what we should see. So having a Robot class desktop screenshot does the trick.

5.3 How scenario playing works

Now that you know all about the way recording works it's time to deeply go into how playing scenarios works. No chance you haven't guess JMX Remote API is also implied into this task. A brief graphic will give you all wanted information:



This allows us to play scenarios on distant computers as well as on localhost.

6 Using jDiffChaser

6.1 Configuration

6.1.1 The trick in your application

First you need to create a sort of back-door to the main frame of your application. In a few words, you need one static method in the main class you will compare screens from:

```
public static JFrame getFrame()
```

Another method you can add (it will be used if present, it's optional) is a method indicating what is the running version:

```
public static String getVersion()
```

Once you have the *getFrame()* static method in your application, you can record and play scenarios with it.

6.1.2 The file tree

A typical files tree of jDiffChaser testing is the following you have in the *jDiffChaser\samplestests\sketchbook-sample* directory:

- *thejDiffChaserTestSessionFile.xml*
- *failed* (directory that will contain all differences found)
- *scenarios*
  - *setup800x600*
    - *scenarioSetup1.sc*
  - *setup1280x1024*
    - *scenarioSetup2.sc*
    - *scenarioSetup3.sc*
  - *tearDown800x600*
    - *tearDown1.sc*
  - *tearDown1280x1024*
  - *groupOfScenario1*
    - *scenarioA.sc*
    - *scenarioB.sc*
  - *groupOfScenario2*
    - *scenarioC.sc*
    - *scenarioD.sc*
  - *groupOfScenario3*
    - *scenarioE.sc*

Note that the *failed* directory will contain directories created by jDiffChaser

for each scenario that will find some differences. It will also contain the *index.html* file of the report.

6.1.3 The classpath

When recording or playing scenarios using jDiffChaser, you need a classpath containing your application classpath as well as the jDiffChaser one.

6.1.4 The tests session file

We already have explored the *tests-sketchbook-sample-win.xml* in our **Quick Start** section. Now let's dig into a more complicated configuration file based on a parallel mode testing environment.

```
<test-configuration>
	<report-title>Tower Controller Positions</report-title>
	<first-host>
		<ip>xxx.xxx.xxx.xx1</ip>
		<port>1202</port>
		<arg name="jndiconf">-conf /conf/twr1/</arg>
	</first-host>
	<second-host>
		<ip>xxx.xxx.xxx.xx2</ip>
		<port>1202</port>
		<arg name="jndiconf">-conf /conf/twr2/</arg>
	</second-host>

	<scenarii-base-directory>testsdata/scenarios/</scenarii-base-directory>
	<failed-base-directory>testsdata/failed/</failed-base-directory>

	<!-- first the clients tests with 1280x1024 resolution-->
	<test-suite parallel-mode="true">
	    <setup-scenarii-directory>setup1280x1024</setup-scenarii-directory>
	    <teardown-scenarii-directory>tearDown1280x1024</teardown-scenarii-directory>

		<gui-test>
			<scenarii-directory>groupOfScenario1</scenarii-directory>
			<arg name="jndiconf">position1</arg>
			<arg name="width">-w 1280</arg>
			<arg name="height">-h 1024</arg>
		</gui-test>

		<gui-test>
			<scenarii-directory>groupOfScenario2</scenarii-directory>
			<arg name="jndiconf">position2</arg>
			<arg name="width">-w 1280</arg>
			<arg name="height">-h 1024</arg>
		</gui-test>
	</test-suite>

	 <!-- then another client test with 800x600 resolution-->
	<test-suite parallel-mode="true">
	    <setup-scenarii-directory>setup800x600</setup-scenarii-directory>
	    <teardown-scenarii-directory>tearDown800x600</teardown-scenarii-directory>

		<gui-test>
			<scenarii-directory>groupOfScenario3</scenarii-directory>
			<arg name="jndiconf">position3</arg>
			<arg name="width">-w 800</arg>
			<arg name="height">-h 600</arg>
		</gui-test>
    </test-suite>
</test-configuration>
```

First, note that all paths are described using the '/' character as separator, even for

Windows system tests.

We find some keywords we didn't encounter in our Quick Start example:

- The `<test-suite parallel-mode="true">` means that scenarios found in the dedicated directories of this test suite will be sent and executed at the same time on both test hosts

- In `<first-host>` and `<second-host>` tags, `<arg name="jndiconf">-conf /conf/twr1/</arg>` : The tested application takes an application argument we give `"jndiconf"` as a key in our configuration file. Part of its content is `"-conf /conf/twr1/"`. The second part of its content is given in each `<gui-test>` tag as, for example, `<arg name="jndiconf">position1</arg>`. That means that, when gui tests of `groupOfScenario1` are launched, the application is given the following parameter: `-conf /conf/twr1/position1`.

- Other arguments are not split, consequently you only find them in the `<gui-test>` tag. For example, the `<arg name="width">-w 800</arg>` argument.

- The `<setup-scenarii-directory>` allows users to give a set of scenarios to play before all scenarios found in the test suite

- The `<teardown-scenarii-directory>` allows users to give a set of scenarios to play after all scenarios found in the test suite

6.2 More on recording

We've seen most of the recording aspects in the **Quick Start** section. But there's one more feature I must explain, the check box with following label: “Next actions must be played on not both tested hosts” you can find on the Remote Control.

When checked (default value), this option makes the scenario send following actions to both hosts. That means, if a mouse move is done on host A, the same move is done on host B. While when it is unchecked, the next actions are only sent to host A, leaving host B in its current state, until you check back this option.

When is such an option useful? It must be used only in parallel mode. We use this, for example, in the following case. You have a text field with a ok button beside. When the button is pressed, the value is sent to all hosts, all hosts have edit access on it. When playing the scenario, we want to test that once the data is sent, all hosts have the same value in this text field, nothing more. We do this by recording actions in “both hosts mode” until we reach the button, then we uncheck the option to enter “single host mode”, we change the text field value and send it with ok. Then we come back to the “both hosts mode” by checking back the option and we take the screenshot, and so on...

Note that such a scenario, when played locally (to test the scenario validity), will play all actions.

7 Tips and Tricks

7.1 Best practices

1. Record really short sessions: it's easier to replace a short recorded session by another. Split a functional set of actions into many little scenarios with file name numbered. They will be played according to the alphabetical order. Keep an eye on the timer when recording a session.
2. Always test your scenario locally (use the local player) before deploying it and using it during night tests sessions.
3. Clean your environment with recorded actions, if needed:
    - at the end of the scenario (you can still record some actions after having taken the screenshot)
    - at the end of the gui-test, in a little dedicated scenario

7.2 MacOS tricks

MacOS Look & Feel is “a bit” ;) different from the Windows one, so jDiffChaser usage is consequently modified a little bit. The main difference that impacts its usage is the frame focus handling. Let's detail some major points you need to know before using jDiffChaser on OSX:

- When switching between the Remote Control frame and the application frame, you have to click on the frame border to gain focus before performing any action on the corresponding interface
- You won't be able to pull down a menu and click on a capture button to test menu entries: as soon as you have clicked on the button, the application frame loses its focus and the menu is closed (hiding the entries you wanted to capture). The way to do menu screenshots using OS X is to define a capture delay in the dedicated textfield (on the remote control frame), click on a capture button (either the fullscreen or application one), then open the menu you want to capture and wait the screenshot to be done (look at the textfield to see the remaining time before the capture)

8 Known limitations (v0.8)

Sometimes you have to wait n seconds before taking a screenshot (when recording a scenario) to be sure to have a correct complete and valid screenshot. We experienced strange Swing Rendering Thread behaviors without this delay for few test cases. We are working on finding a quicker/better solution.

About fullscreen screenshots you can take with this version: beware that such tests depend on your screen resolution (when choosing the parts you don't want to compare, those zones will change with the resolution set on the tested host). That's why we don't provide a window move test scenario in our sample test (empty directory). Just record your own and test it ;)

9 Improvements backlog

- Blinking is hard to visually compare, we are thinking about a way to “view it”...
- Find a way to record user actions without having to code a public access to the main frame of the application
- Regression can sometimes only imply to keep some colors or text or... We should add some regression assertions in jDiffChaser: e.g. “This button should have a red dominant color”, “This button is blinking between green and black every 1 sec.” and so on... But we need to be very careful about having a limited scope dedicated to regression assertions, not extending to functional assertions. This may be another project ?
- Record a scenario using a storyboard provided by the interface design team
- Choose to define zones to compare instead of defining zones to ignore (when taking fullscreen screenshots)

10 Changelog

10.1 Version 0.8

- Full screen capture (allows multiple window applications to be tested, window moves to be tested,...)
- Delay before screenshot (if needed, 0 sec. delay is the default one)
- No need of bsh anymore to build jDiffChaser using ant
- Fix of a focus bug when having the end-of-recording dialog displaying
- Some waiting dialogs added in order to give some better UI feedback to the user.
- Frames and dialogs moves are handled
- Native libraries are only used when using java < 1.5 on Windows in order to have always on top dialogs (Remote Control frame and waiting dialogs). Consequently, OS X and probably other java-enabled platforms are now supported through java 1.5+ (Note that we need feedback from Linux users as we still didn't do some tests with it)
- Report details: when clicking on an image, now allows to browse the three images of the scenario (first one, second one and diffs one) with previous/next buttons
- Needs jdk 1.5+ to build a jar file that you can use either on hosts running Windows with 1.4.2 java or any OS with 1.5+ java
- Fixes an OSX application restart bug that occurred when playing gui scenarios suites
- Remote Control frame can be translucent: useful when recording full screen applications scenarios