JavaStar User's Guide

THE NETWORK IS THE COMPUTER™

SunTest, Inc.
A Sun Microsystems, Inc. Business
901 San Antonio Road
Palo Alto, CA 94303 CA USA

Contents

About JavaStar 1

JavaStar™ is a professional testing tool designed for Java users who want to test their applications thoroughly and effectively. JavaStar tests your program Graphical User Interfaces (GUIs) using the power of Java, accessing the attributes of GUI components and verifying their accuracy.

The JavaStar application has many functions and options, all supporting the primary features that allow you to:

- Record user interaction (mouse events, keystrokes) to Java scripts, and define comparisons of GUI components during the process.
- Compose scripts into more complex tests using sophisticated controls.
- Playback scripts and tests in realtime, step-by-step, or with modified time delays
- Enhance existing JavaStar scripts or create your own with the aid of the JavaStar test libraries

This chapter describes:

Benefits of Using Java to Test Java

JavaStar provides many benefits simply because the application, as well as all the JavaStar libraries and the code it generates, are all Java. This means that:

- JavaStar and its tests are portable across machines—you can run the program and your tests in any environment that supports Java, without needing to modify your scripts.
- The JavaStar record and playback processes "live" in the same address space as your test application or applet. This gives JavaStar full access to the attribute information of the program's GUI components.

- As a Java program, JavaStar can get access to the attributes of your application or applet's GUI components, allowing you to test your program in more depth.
- You can expand and enhance JavaStar scripts using Java. You don't need to learn another proprietary language to begin working productively.

Recording Scripts

JavaStar's Record function is simple and straightforward. You specify the application or applet you want to test, define the name of the script, and then start using your test program in the way you want to test it. JavaStar records all events (and, if you choose, the exact delays between events) to a script. The script is a .java file that JavaStar compiles into a .class file that you can later run.

Because you're testing a GUI, the state of your program's components, as well as their look on screen, are all important to validation. JavaStar addresses this by providing two comparison-based features: Verify and Synchronize. These gather master comparison data and insert comparison checks into your code during recording, then use this information to evaluate matches during playback.

Verify can compare the contents, attributes, and enabled state of any GUI component. It can also compare the image of the component if Java allows images for that particular item. If, during playback, a verification fails, the JavaStar notes the verification in the log file and continues on with the script.

Synchronize does a similar comparison, but if the results don't match, the test script ends with an exception. This gives you a way to detect inappropriate state changes that threaten the integrity of the rest of the test.

Composing Tests from Scripts

In the JavaStar model, you create small, re-usable *scripts* that address specific functionality and join these together to compose complex *tests*.

While it might, at times, seem easier to record a complete test in one script, start to finish, this approach limits your ability to expand and maintain the script. The script can't be reused for other purposes, and the length and complexity makes it difficult to keep up to date.

With the modular script approach, the scripts you write are focused and compact. For example, you might write one script that tests a specific dialog window and nothing else. You can use this script in any test where you bring up that same dialog window. If the window changes, you only have to change this one script. And if you decide you want to improve the testing within the script, your changes immediately update all tests that call this script.

Of course, this concept isn't new; it's the foundation of object-oriented programming. What's different is that JavaStar is built around this model, and includes features that make the implementation as efficient as possible.

Composing Tests in JavaStar

In JavaStar, you combine scripts into tests using the Compose Test feature. In Compose view, every script is a node in a test tree that you design. The tree is displayed graphically, making it easy to move nodes around and define their attributes. You define the relationships between nodes—whether their dependency is based on a normal condition or an exception—to define the test flow based on results. This provides you with a way to recover from (but still capture) failures that might otherwise halt automated testing and waste time.

You can also define whether a node is a restart node (restarting the test application or applet) and you can pass parameters to any node with a script that accepts them.

An example of a composed test

As an example, Figure 1-1 shows a simple test composed using five scripts. These tests execute against a small name database applet that requires data to reveal some of the GUI components. Each script in the test represent an discrete tasks—one that might well be needed by other tests for this applet.

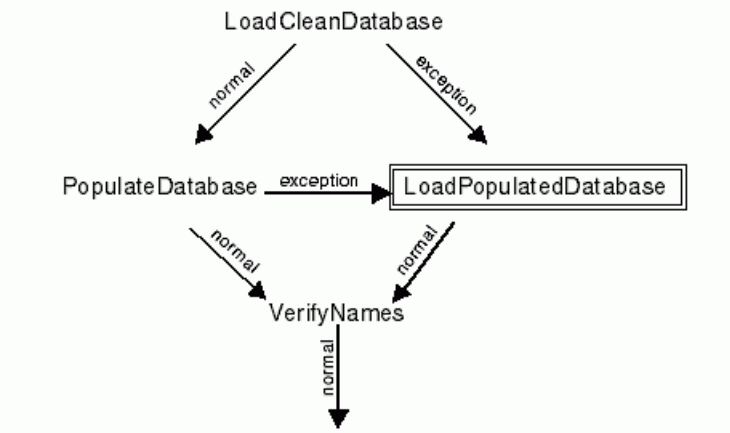

Figure 1-1 Scripts joined together to form a single test

In this composition:

1. The `LoadCleanDatabase` node (the root node) starts the test applet. This happens without any direction from the user.
2. `LoadCleanDatabase` prepares the test environment by loading an empty database in to the applet, then checks to see if the database loads correctly.
3. If the database loaded correctly, the script ends normally and continues on to PopulateDatabase.
4. If, however, `LoadCleanDatabase` throws an exception, the `LoadPopulatedDatabase` script executes instead, restarting the application and loading a populated database, thus recovering the situation.
5. If `PopulateDatabase` can't finish normally, it also calls `LoadPopulatedDatabase` as a recovery mechanism.
6. When `PopulateDatabase` or `LoadPopulatedDatabase` complete normally, the `VerifyNames` script executes next.

In this small test sample, you can see how the reusable features of scripts and the controls available when composing tests can come in handy. With carefully planned tests, you can write test suites that execute for longer periods of time and provide you with more information. You can build new tests out of similar components. And, if you enhance a script (or write your own from scratch) using Java, you can pass parameters to that script as part of your test. This moves your testing beyond simple record-and-playback scenarios.

Playing Back Scripts and Tests

You can play back JavaStar scripts and tests using varying levels of automation. For developing scripts, you can load and playback scripts while preparing to record other scripts. When you have a composed test, or a script that can function without the aid of any other script, you can run this from the command line, or using the Run Test feature.

When you play back scripts, you can reset the delay and timeout values to tailor them to a specific need.

Delay values are used in scripts where you either recorded delays as part of the test (this is an option) or where you inserted a call to the delay method of the JavaStar library. At playback time, you can give a new value that JavaStar multiplies by the existing delays to provide a new timing scenario.

Timeout values are used in verify comparisons. After failure, a verify comparisons (unlike a synchronization) repeats until it reaches the timeout value. At playback, that might be a variable you want to change.

Because you might be running a test suite using a test harness, both the command line playback mode and Run Test give you control on whether the JavaStar GUIs and your test program display during execution.

Providing Access to Java's Full Power

If you know Java, you can take JavaStar scripts to a higher level and add even more power to your test effort. To support Java-savvy users, JavaStar provides ways to:

- Tailor your scripts to more demanding scenarios by adding advanced scripting and comparison capabilities from the JavaStar Application Programming Interface (API).
- Quickly generate Java classes containing the GUI component declarations for your application or applet. You can then write new tests or modify existing tests to use the component map and leverage the power of an abstracted GUI interface. When the interface changes, you change the map, and your tests continue to run properly.
- Use Java tests written without JavaStar, or JavaStar-generated tests that you have modified to add custom Java code.
- Pass parameters to more sophisticated tests to create modules that are even more usable (for example, a file call where the filename is a parameter specific to a composed test).
- Edit and compile scripts on-the-fly, without needing to run another application.

Additional Features

In addition to features that create and execute tests, JavaStar also gives you ways to:

- Use filters on test log files to see only the result data relevant to you.
- Run multiple processes—recording, playback, and so on—simultaneously.
- Monitor job status and terminate jobs when test programs hang.

1

Preparing to Use JavaStar 2

This chapter covers the basics you need to know to set up your JavaStar environment and begin using the product.

Here you'll find:

- Instructions for Installing JavaStar
- Instructions on Starting JavaStar
- An introduction to the JavaStar Main Menu options, including pointers to where these options are explained in the User Guide
- Guidelines for Creating a Project File
- Instructions on Starting Your Application or Applet within JavaStar

Installing JavaStar

On a UNIX System

1. **Download the UNIX version of JavaStar into a directory on your local system.**

2. **Uncompress `javastar.tar.Z` by typing:**

   ```
   uncompress javastar-117.tar.Z
   ```

3. **Type the following command to unpack the `javastar-117.tar` file.**

   ```
   tar -xpf javastar-117.tar
   ```

   This installs JavaStar into a directory called `javastar`.

4. **Set the CLASSPATH variable to include the JavaStar path as the first item.**

   These examples illustrate setting the classpath in a Solaris environment:

   ```
   setenv CLASSPATH /JavaStarPath/javastar/javastar.zip
   ```

   where JavaStarPath is the parent of your `javastar` directory.

   If you already have the `CLASSPATH` variable defined, add JavaStar to the beginning of the path.

```
setenv CLASSPATH
/JavaStarPath/javastar/javastar.zip:$CLASSPATH
```

Under Windows 95 or Windows NT

1. **Download `JavaStar.exe` to a local drive.**
2. **Double-click on `JavaStar.exe` to begin InstallShield.**
   The installation program now guides you through the setup process. The installation automatically sets the `CLASSPATH` variable for you.

Starting JavaStar

On a UNIX System

To start JavaStar from other environments, such as UNIX, you first need to set the CLASSPATH variable to include the JavaStar application path as the first item. To do this:

1. **Type:**

   ```
   set CLASSPATH=JavaStarPath/javastar.zip
   ```

   Where JavaStarPath is the path to the directory where you installed JavaStar.

2. **Type:**

   JDKpath`/bin/java javastar`

   Where *JDKpath* is the path to the JDK directory.

Under Windows 95 or Windows NT

To start the JavaStar application in Windows 95 or Windows NT:

- **Double-click on the JavaStar icon.**

Depending on the choices you made during installation, you'll find the JavaStar icon on the desktop or in the Suntest program group.

JavaStar Main Menu

Once you launch JavaStar, the main menu appears, along with the Project Settings screen:

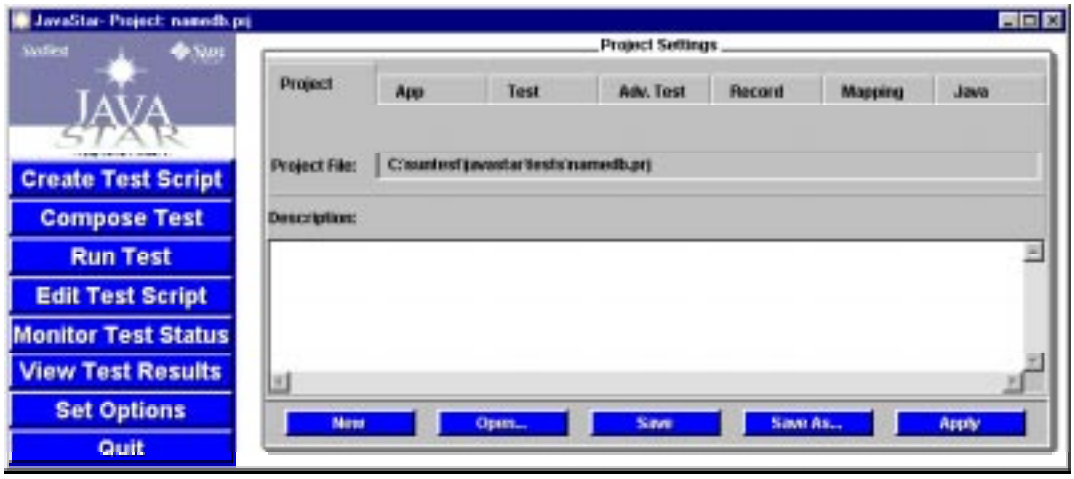

Figure 2-1 JavaStar main menu

Create Test Script is where you start your application. This leads you to the Record / Playback window where you can record scripts, generate declarations, play back scripts (primarily for debugging), and inspect the application components without recording your actions. These functions are described in different chapters—see the chapters "Recording Scripts," "Interacting and Inspecting," "Generating and Using Declarations," and to some extent in "Running Tests."

Compose Test is where you combine scripts to form complex tests: adding pass / fail dependencies, passing parameters, setting scripts to restart the application, and more. You can find instructions on how to compose tests in the chapter "Composing Tests."

Run Test is where you run tests that log results. In this window, you can specify arguments to send to the test, and override your default directory settings. This option, as well as other ways of running tests, is covered in the chapter "Running Tests."

Edit Test Script is where you edit and compile scripts and declaration files. Read about editing features in the "Customizing Options" chapter.

Monitor Test Status is where you track the status of multiple JavaStar processes. You can terminate execution of a process from here. The chapter "Monitoring Test Status" describes how this works.

View Test Results opens the full-featured Results Viewer. Here you can view the results of any test run, getting summary or detailed information, filtering for the results you want. You can also archive, extract, or print results from here. See the chapter "Viewing and Analyzing Results" for details on how to use the Results Viewer.

Set Options is where you can view system information and set options that control the look of the user interface. For more information, see the chapter "Customizing Options."

Quit exits the JavaStar application.

The **Project Files** panel is where you define the application or applet you want to test and the JavaStar options you want to use while testing that program. See the section "Creating a Project File."

Creating a Project File

JavaStar uses project files to store information about directory locations, classpaths, the program to test, and various record and playback options.

A project file is usually specific to an application or applet, and to a platform. The tests you create with JavaStar are platform-independent; by using project files to define platform-specific information (such as the location of your Java Virtual Machine) you can set your test environment in one place, rather than having to define it for each test run.

If you previously used the `javastar.prop` file to store your options, JavaStar will prompt you to convert your file when you run this version.

For instructions on defining or modifying a project file, see the chapter "Creating Project Files."

Starting Your Application or Applet

Before you can create test scripts or generate declarations for your program, you need to start up the application or applet you plan to test. If you included this information in the currently loaded project file, you don't need to provide it again. You can always override your project file settings for a session if you choose.

The settings you enter into the Create Test Script dialog window stay in effect while the Record / Playback window is open and the application or applet is running. These settings are not saved to the project file; use the Project Settings screen to save changes to the file.

1. **From the main menu, click** Create script**.**
   This brings up the Create Test Script dialog window.

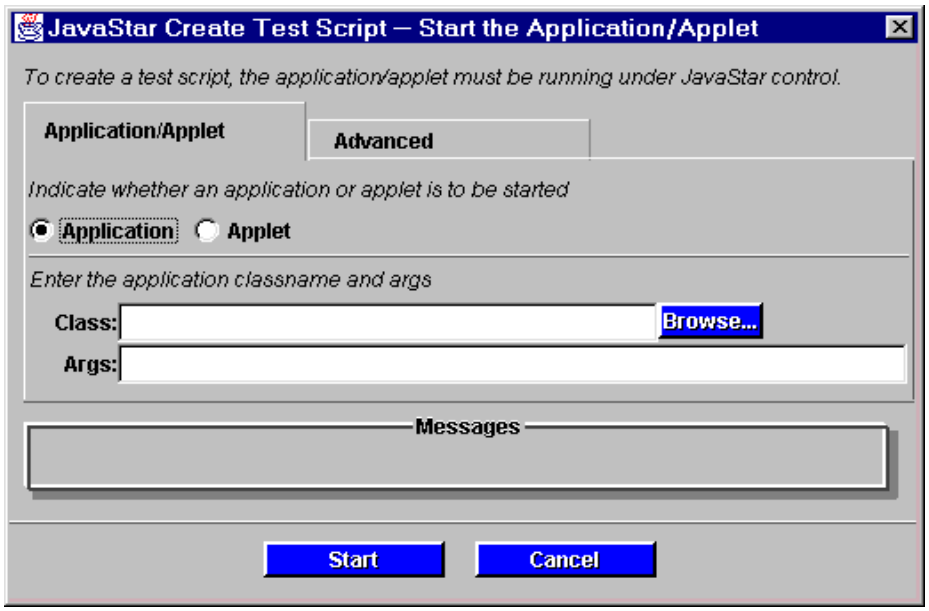

Figure 2-2 Create Test Script dialog—showing Application fields

This window has two tabs: Application / Applet and Advanced. Use the Application / Applet tab to specify the information specific to your test program. Use the Advanced tab to specify the path to the program (if it is not already defined in your CLASSPATH setting) and to change directory settings from your defaults.

2. **Specify the type of program you are running: Application or Applet.** The fields shown on this tab change depending on your choice. When you click on the Applet checkbox, the screen changes to show new fields. See Figure 2-3.

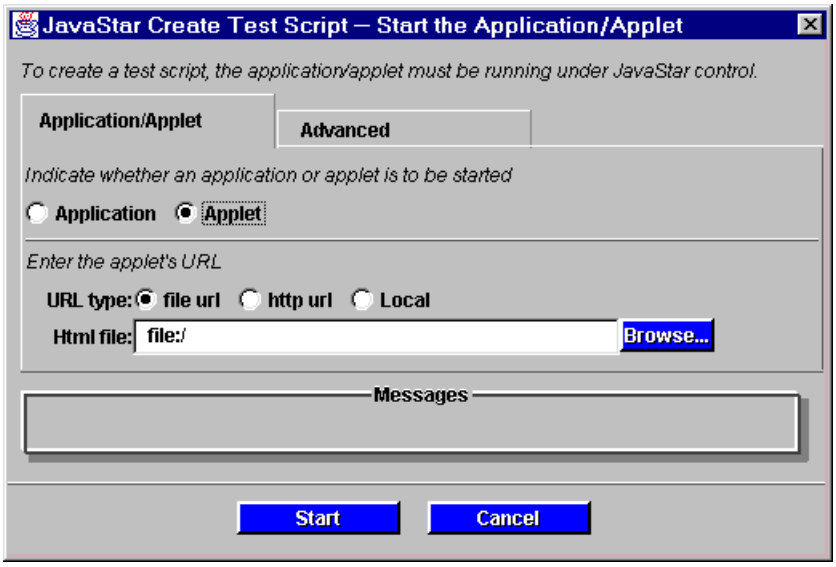

Figure 2-3 Create Test Script Dialog—showing Applet fields

3. **Fill in the fields as appropriate to your test program.**
   For an application, you need to define:

Table 2-1 Application information fields

Field	Description
Class	The class name of the Java application you want to test. Either enter the name (without the `.class` extension) into the text field, or use the Browse button to locate the application by navigating through the directory structure. Make sure you use a fully-qualified class name—for example, if your class is part of a package, specify the package here, too. For example, if your application is in a package named `beta` and the name of the main class is `dialer.class`, you would type `beta.dialer` into this field.
Args	Any arguments you want to pass to the application under test.

For an applet, you need to define:

Table 2-2 Applet information fields

Field	Description
URL type	Choose between file url, http url, and Local. • Choose file url on Windows systems to specify a local directory and filename for the `.html` page that runs your applet. • Choose http url when you need to run an applet remotely across the web. • Choose Local when you need to specify a local directory path and filename for the `.html` page that runs your applet. The local option works best in UNIX environments; use file url for Windows environments.
HTML file	The location of the web page containing the Java applet you want to test. Either enter the location (using a prefix) into the text field, or use the Browse button to locate the file by navigating through the directory structure.

4. **Check the Advanced settings to make sure they are correct for the script you want to create.**
   You can skip this step if you are already sure you test program is in the CLASSPATH and that your directory defaults (as specified in Environment options) are appropriate for the script you want to create.

   To check these settings, click the Advanced tab. The screen changes to show advanced options.

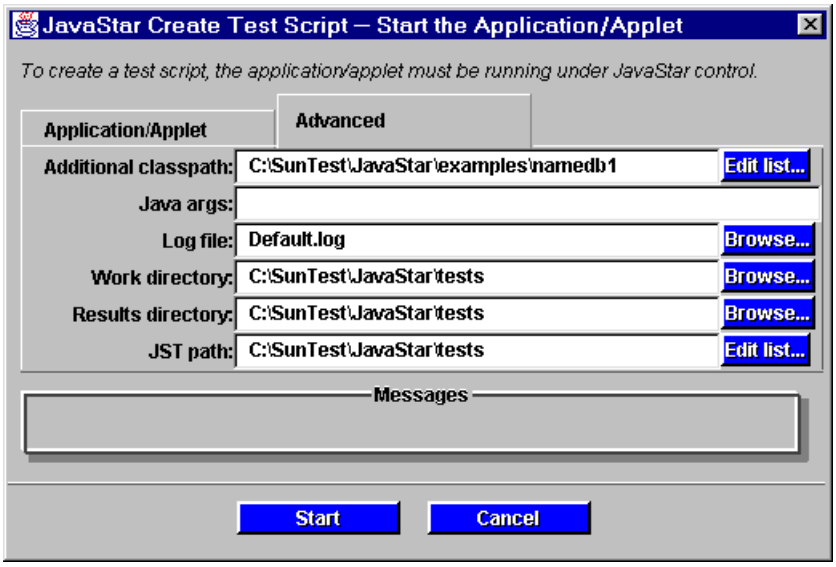

Figure 2-4 Advanced tab on the Create Test Script dialog

You can optionally define or change any of these fields:

Field	Description
Additional classpath	Any additional directories, `.zip`, or `.jar` you want to add onto the existing `CLASSPATH` variable definition. This must include the path to any applications or applets you want to test, tests scripts you want to execute, and any map files you want to use. When specifying a `CLASSPATH`, use a semi-colon for a separator if you are running under Windows, and use a colon for a separator on UNIX platforms.
Java arguments	Any default flags for either JVM or the Java compiler that your program under test require.
Log file	The filename JavaStar creates for writing out results. By default, JavaStar uses the filename `Default.log`.
Work directory	The directory where JavaStar stores the scripts, result logs, and comparison directories your scripts generate.
Results directory	The directory where JavaStar creates the gold file comparisons and results directory.
JST path	The directories you want JavaStar to search when finding the scripts and JSTs necessary to run a JST. When listing multiple directories, use a semi-colon for a separator if you are running under Windows, and use a colon for a separator on UNIX platforms.

5. **Click** Start.
   JavaStar launches your program and opens the JavaStar Record / Playback window (shown in Figure 2-5.) From here, you can playback a script, record a new script, interact with the application without recording, and generate declarations for your GUI.

   If JavaStar puts up an error message instead of launching the program you want to test, see If Your Application Does Not Start for ways to resolve the problem.

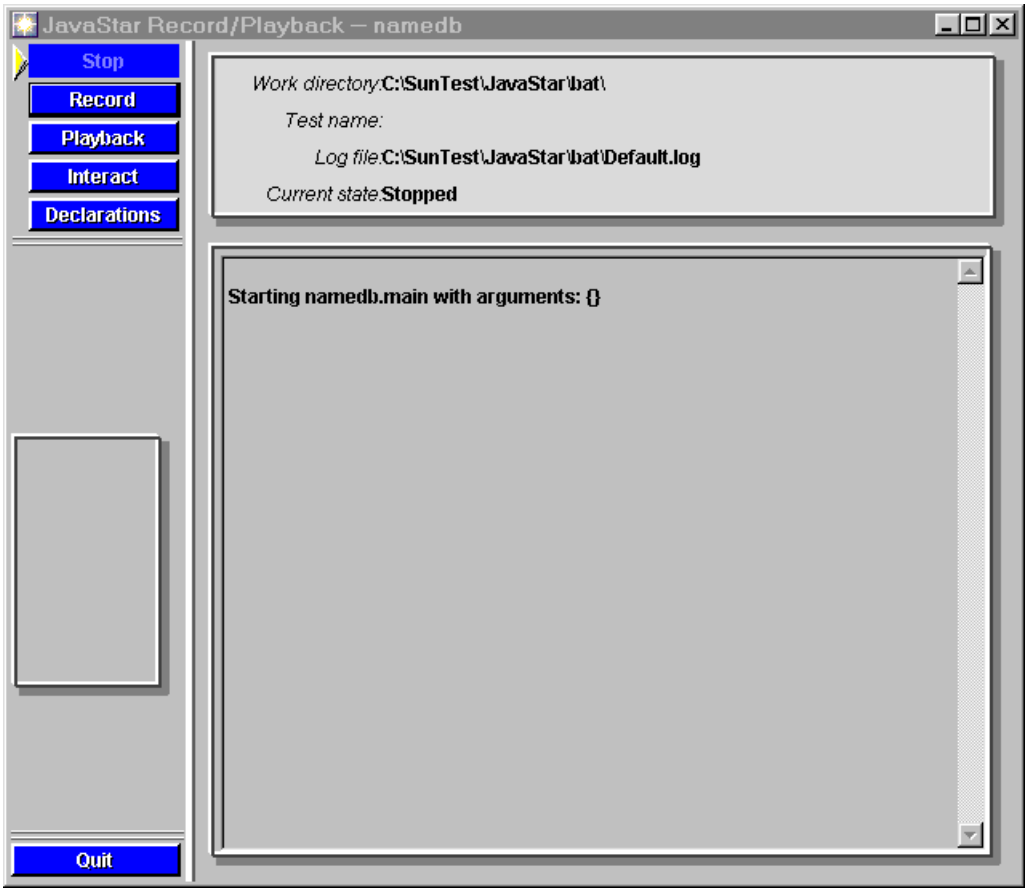

Figure 2-5 Record / Playback window.

If Your Application Does Not Start

Here are some possible error messages you might get, and how to resolve them:

```
There is some problem accessing the class classname. Either:
1. It is not in the CLASSPATH.
2. It is being accessed in the wrong way.
(e.g. String is invalid, java.lang.String is correct.)
```

Possible solutions:

- Make sure your application path is included in the additional classpath field. Check the entry for this field in the Advanced tab. If you typed the path in directly, you might try using the Browse button to navigate to the directory, so you can be sure you have the latest path.
- Check that you spelled the application name correctly in the Class field of the Application/Applet tab. Do not include the `.class` extension. Make sure this class is the "main" class of your application. Use Browse to navigate to the directory and make sure the `.class` file is still there.
- Verify that you are using a fully-qualified class name. If your class is within a package, be sure to type packageName.className.

2

Creating Project Files 3

JavaStar uses project files to store information specific to a project or to a platform. In a project file, you store information such as the name of the program under test, directory settings, CLASSPATH additions, and record / playback options. Defining a project is the first step in setting up JavaStar for a given project.

This chapter covers:

Understanding the Project Settings Window

In a project file, you specify:

- Information required to run the application or applet under test
- CLASSPATH information
- Which directories to use to find tests and to generate output
- Option settings that control recording and playback
- Java settings (virtual machine and compiler information)
- Mapping information for locators, declaration files, and text maps
- A description of the project file

When you open JavaStar, the last project file you worked with is automatically loaded (see Figure 3-1). If you have not yet saved a project file, JavaStar loads a default file named `default.jpr` (where `.jpr` indicates a **J**avaStar **PR**oject file).

Project files are especially important when testing a project across different platforms. The Java Virtual Machine (JVM) settings and file paths will vary across platforms; by setting up a project file for each platform, you only need to define these paths once.

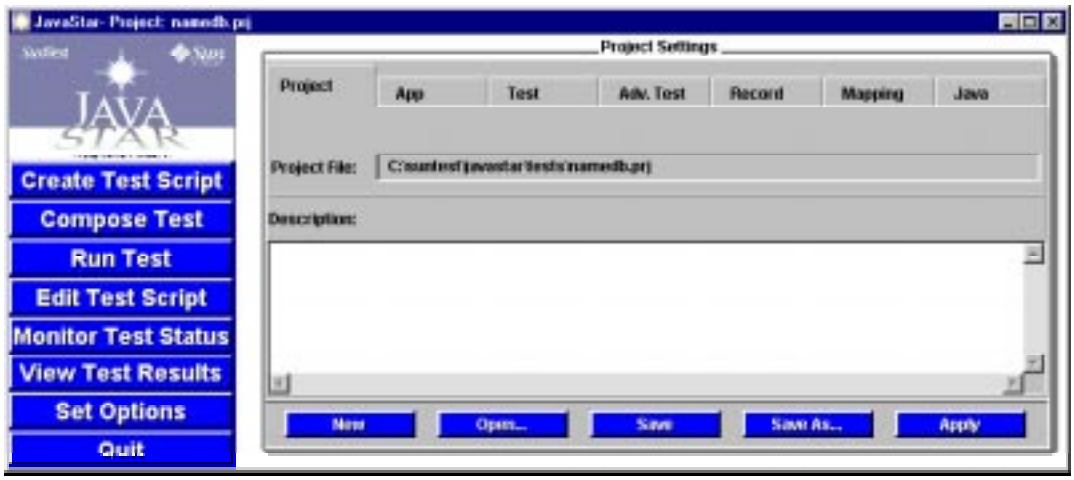

Figure 3-1 Main menu and Project Settings panel

Project Settings contains a series of tabbed panels. Clicking on a tab along the top of the window brings the associated panel to the forefront.

Defining Project Information

When you open JavaStar, the Project tab of Project Settings is displayed to the forefront by default (see Figure 3-2). This panel shows you the name of the project file and has an editable text area—Description—where you can describe the purpose of the file.

> ***Note –*** The project file name is shown as a display-only field. To change the filename, you need to use the Save As... button. See the section "Saving, Applying, and Loading Project Files."

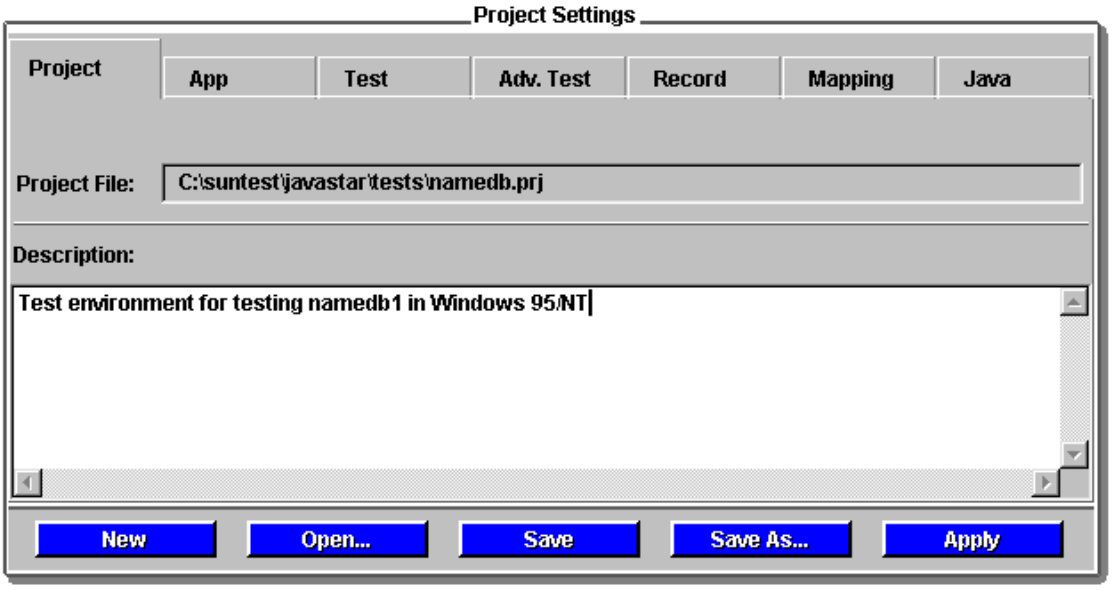

Figure 3-2 Project panel

By clicking on the tabs along the top of the Project Settings screen, you can bring other panels to the forefront. This section describes:

- Providing Application or Applet Information
- Setting Test Options
- Selecting Record Options
- Recording Format Options
- Specifying Java Options
- Defining Locators, Declaration Classes, and Text Map Classes
- Advanced Test Options

Providing Application or Applet Information

The App panel (Figure 3-3) is where you provide information about the program you want to test. For applications, you need to provide the name of the class file, any arguments the application requires, and the addendum to the CLASSPATH variable that JavaStar will use to locate the application. For applets, you need to provide the name of the HTML file that runs the applet.

To record or run a test, you do not have to provide this information in the project file. However, you can save yourself a lot of retyping if you enter the information here. If you will always be using the same application or applet in conjunction with this project file, or even if you will be using a particular program most of the time, go ahead and fill in the information here. When you later choose Create Test Script from the main menu, you'll have an opportunity to override these settings for that record / playback session.

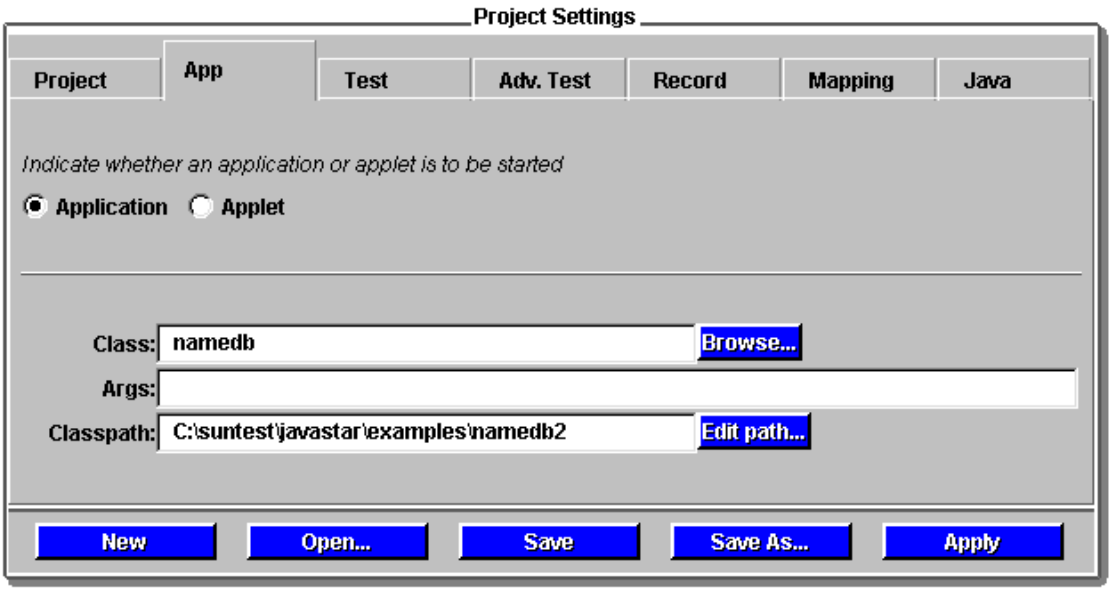

Figure 3-3 App panel

To fill in application or applet information:

1. **Select the type of program you plan to test.**
   Click either the Application or Applet radio button.

2. **If you are testing an application:**

   a. **In the** Class **field, type the name of the class to test.**
   You can also use the Browse button to locate the class by navigating though the file dialog. If you type the name, you don't have to provide the `.class` extension.

   When you select the class name using the Browse button, JavaStar automatically enters the path to that class in the Classpath field.

   b. **In the** Args **field, type any arguments you need to pass to the application.**
   Enter these exactly as you would at the command line. If your application does not require arguments, leave this field blank.

3. **If you are testing an applet, enter the name of the html file for the applet.**
   You can type the name directly into the Html file field, or use the Browse... button to locate the file using the file dialog.

Setting Test Options

The environment preferences affect the overall test environment. In the Test panel (see Figure 3-4) you define the directories you want JavaStar to use when generating tests and test results. You also define additional CLASSPATH settings, time out values, and delay values.

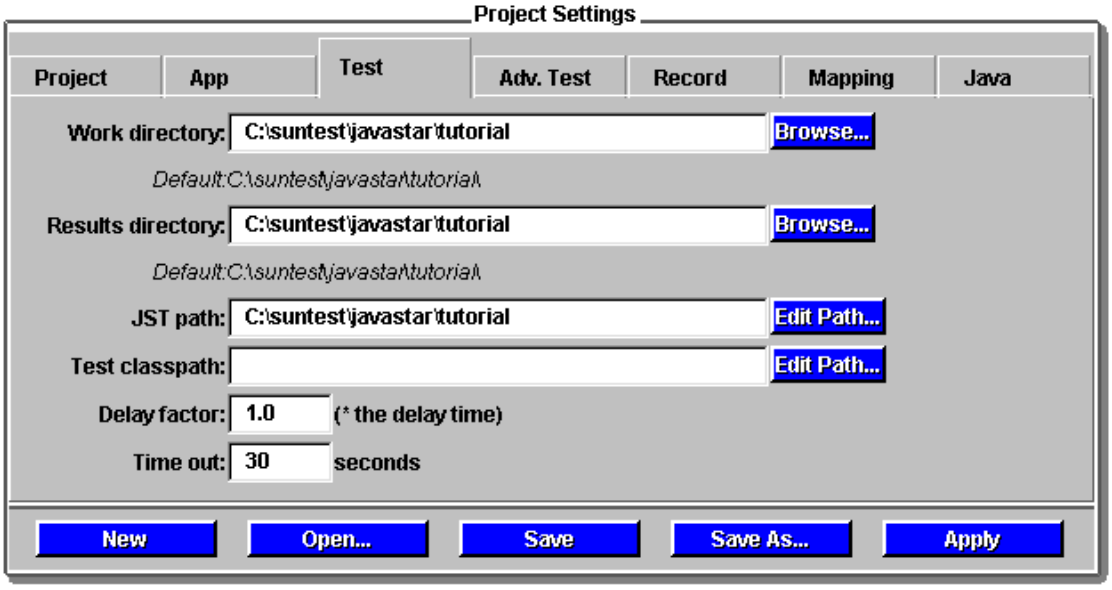

Figure 3-4 Test panel

Note – You can provide additional test options in the Adv. Test (Advanced Test) tab.

If you routinely store your tests in a directory other than the one where you start up JavaStar, type your test directory path in the Work directory field.

Field options:

Work directory	The default directory where JavaStar stores your JSTs, scripts, and gold directories for those scripts.
Results directory	The default directory where JavaStar stores the log files any failure data (from failed comparisons), as generated by your script.
JST path	The directories you want JavaStar to search when finding the JSTs necessary to run a JST.

Test classpath	The path to the directory containing the tests you want to run.
Time out	The timeout value JavaStar uses when verifying comparisons and other `JSComponent` operations. When an operation fails, repeats its attempts until the timeout value is reached. By default, it is 30 seconds, but you can change it here.
Delay factor	This value affects calls to `JS.delay`. If you recorded your script with delays, your script has `JS.delay` calls. When running a test script, JavaStar multiplies the delay factor by the `JS.delay` value defined in the script, and from there calculates the actual delay value. For example, if the value of `JS.delay` is `100` (milliseconds) and the value in the Delay field is `0`, then the actual delay would be 0, but if the value in the Delay field is 2, then the actual delay would be `200` milliseconds. For more information about `JS.delay`, refer to the wrap(Component) description in the JavaStar API Reference.

Selecting Record Options

In the Record panel of the Project Settings window, you can set recording format options and select optional events you want JavaStar to record.

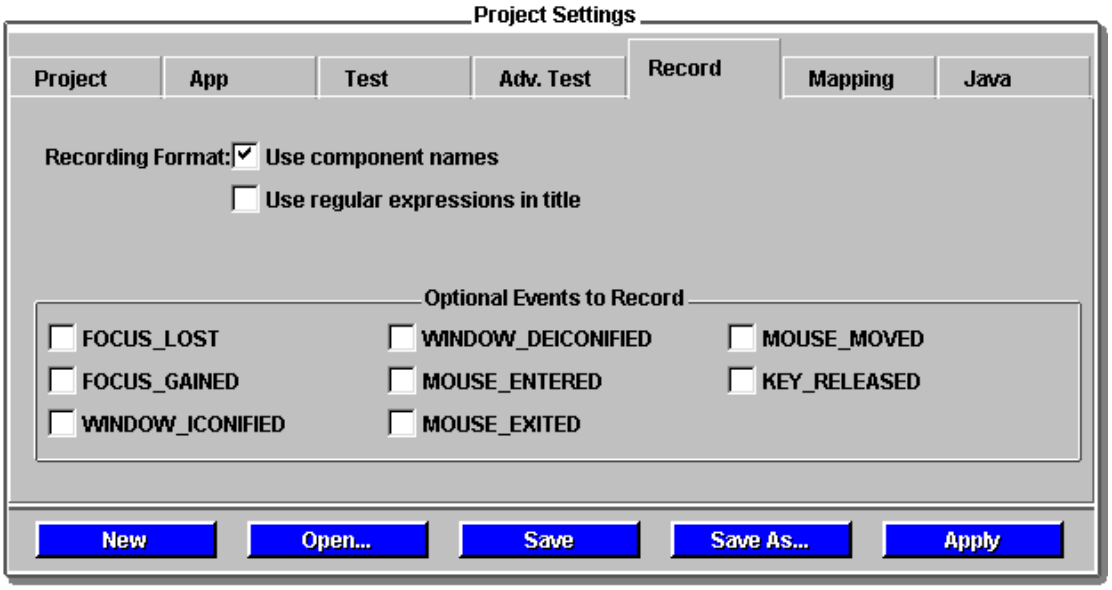

Figure 3-5 Record panel

Recording Format Options

The recording format options define how JavaStar generates code that uses components.

Use component names	Turn this on if you're using `setName()`.
Use regular expressions in title	Automatically records references to frame titles as regular expressions. If your test program has windows that change titles, turning this option on can increase the chances that JavaStar will find the window even with a name change. However, if you have several windows titles starting with the same letter, this might cause a conflict during the test run.

Option Events to Include

Your settings under Optional Events to Include determine whether JavaStar records certain types of events. By default, JavaStar records none of these events on this list. This is useful in cases where your application does not process particular events—for example, the `MOUSE_MOVE`, `MOUSE_ENTER`, and `MOUSE_EXIT` events that are generated every time you move the mouse. If your application *does* process these events, then you should check the check boxes to activate this feature.

The event...	Means the user...
FOCUS_LOST	is not focusing events on this object.
FOCUS_GAINED	is focusing events on this object.
WINDOW_ICONIFIED	minimized the application window to an icon.
WINDOW_DEICONIFIED	restored the application to normal screen view.
MOUSE_ENTERED	moved the mouse *into* the target object.
MOUSE_EXITED	moved the mouse *out of* the target object.
MOUSE_MOVED	is moving the mouse with no buttons pressed.
KEY_RELEASED	finished pressing a key.

Specifying Java Options

The Java panel (see Figure 3-6) in the Project Settings screen is where you define information on the Java virtual machine and the compiler you want JavaStar to use when running and compiling tests. Based on the location of the JDK, JavaStar provides default paths for the JVM and the compiler. If you plan to use these locations, you don't need to fill in the fields. You only need to provide a path if you want to override the default.

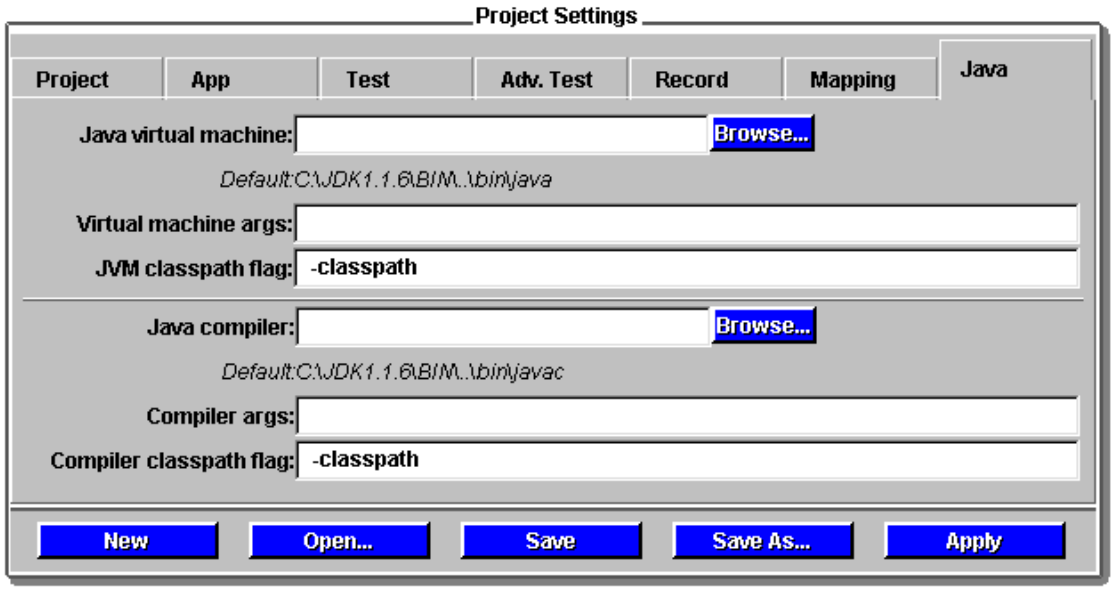

Figure 3-6 Java panel

Java options:

Java virtual machine	The path to the JVM you use. Immediately below this text field, JavaStar displays (in italic) the default path that it will use if you do not specify a JVM.
Virtual machine args	Any Java arguments you want to pass to the JVM. If you want to increase JVM memory, for example, you would use the `-mx` command here.
VM classpath flag	The flag your JVM uses to preface a `CLASSPATH` setting. By default, this is `-classpath`.
Java compiler	The complete path to the compiler you want JavaStar to use. Immediately below this text field, JavaStar displays (in italic) the default path that it will use if you do not specify a Java compiler.
Compiler args	Any arguments you want to pass to the compiler when JavaStar compiles a test.
Compiler Classpath flag	The flag your Java compiler uses to preface a `CLASSPATH` setting passed at the command line. By default, this is `-classpath`.

Defining Locators, Declaration Classes, and Text Map Classes

Use the Mapping panel (see Figure 1-7) to define the names and paths to:

- Non-Component Locators
- Declaration Classes
- Text Map Classes

If you use the Edit List... buttons to select files for these fields, JavaStar will automatically update the Classpath field with the paths to each file you choose. The section, "Using the Selection Dialog" explains how to select files using Edit List.

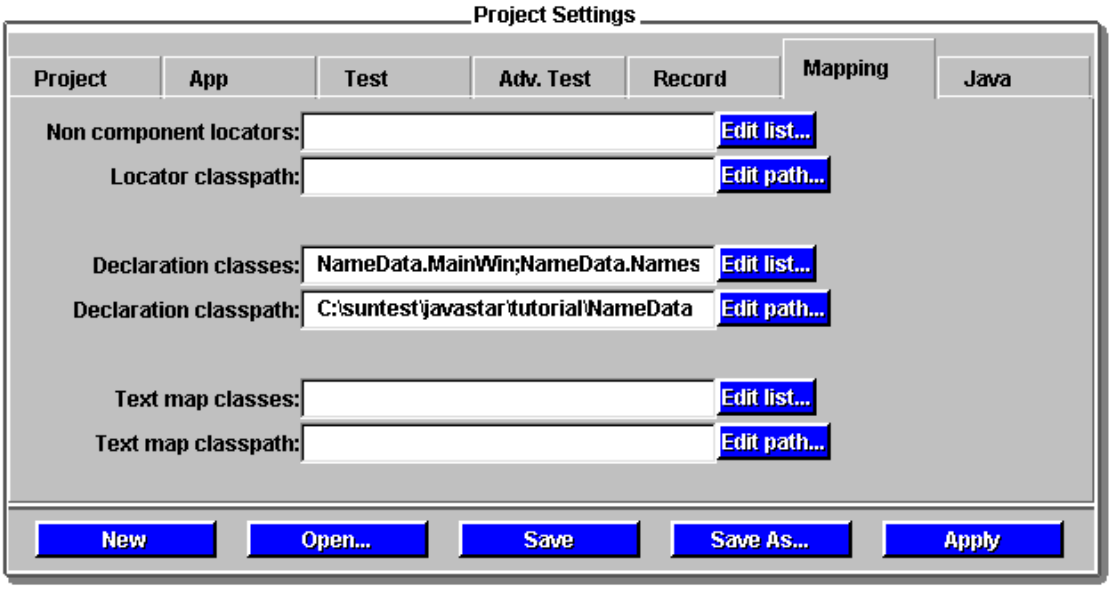

Figure 3-7 Mapping panel

Non-Component Locators

When you test an application created using a toolkit that does not extend `java.awt.Component`, you need to provide JavaStar with a locator. This locator may be one you have written yourself or one provided with JavaStar. A locator is specific to a toolkit—for example, different locators are provided for Bongo, JFC, and IFC toolkits.

If you think your application or applet might need a locator, but you are unfamiliar with the locators available or don't know how to write one, see the chapter "Locators for Non-Components" for basic information. For an example of a locator and how you would use it, see the chapter, "Using Non-Component Locators" in the *JavaStar Tutorial*.

If you already have one or more locators to use with your program, you can use the Non component locators field (editing it with the Edit list... button) to specify the locators and have JavaStar add the location to the class path. For information on how to use the dialog that is displayed when you click Edit list..., see "Using the Selection Dialog."

Declaration Classes

If you plan to use declaration files to abstract GUI components, you need to provide the names of the declaration files and add them to the CLASSPATH environment variable. You can do this in this panel, using the Edit list... button

to the right of the Declaration classes field. After you add declaration classes to the list, JavaStar fills in the Declaration classpath field with the directory paths that correspond to the files you selected.

For information on how to use the dialog that is displayed when you click Edit list..., see "Using the Selection Dialog."

For step-by-step instructions on how to create declaration files, see the chapter, "Generating and Using Declarations" in this *User's Guide*. For a discussion of how to use declaration files within the JavaStar test model, as well as an example, see the chapter "Generating Declarations" in the *JavaStar Tutorial*.

Text Map Classes

Text map classes are utilities you write that map components to text names. If your application or applet uses bit-mapped images for components or if it uses lightweight components of a custom design, you may want to create a text map so that your tests can extract a meaningful name for the component. In cases such as these, a text map can make tests and results easier to interpret.

For information on how to develop a text map, see the chapter, "Text Map Classes."

Using the Selection Dialog

To select files for either the Non component locators, Declaration classes, or Text map classes field:

1. **Click the** Edit list **button to the right of the text entry field.**
   A Select dialog opens. See Figure 3-8.

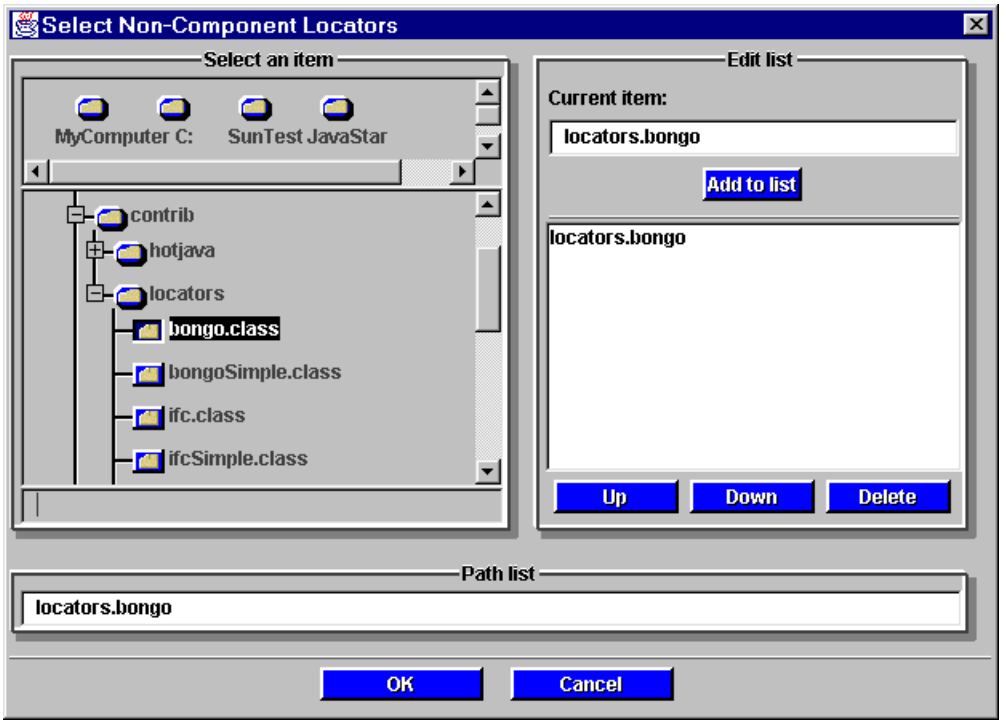

Figure 3-8 Select Dialog (this one is for Non component locators)

2. **In the Select an item panel, navigate to the directory containing the file you want to load, and select the file.**
   For example, if you are selecting a locator, navigate to the `javastar\contrib\locators` directory and select the locator you want.

   The file name appears in the Edit list panel, in the Current Item field.

3. **Add the file to the list by clicking** Add to list**.**
   The file is displayed in the Path list panel and in the list box of Edit list.

4. **Repeat** Step 2 **and** Step 3 **for any additional files you need to add to the list.**
   If you have multiple files listed, you can move then up or down on the list, or delete them, using the buttons at the bottom of the Edit list panel.

5. **Click** OK**.**
   JavaStar automatically enters the name of the class you selected into the text field for this file type, and the path to that class into Classpath field.

You can add additional locators using the Edit list... button.

Advanced Test Options

The Adv. Test panel (Figure 3-9) provides advanced options for controlling tests. Here you can control:

- The amount of time JavaStar waits before it determines that the program under test has hung
- Log output (information types and total size)
- Whether JavaStar reloads classes before running a test
- Prefixes to ignore

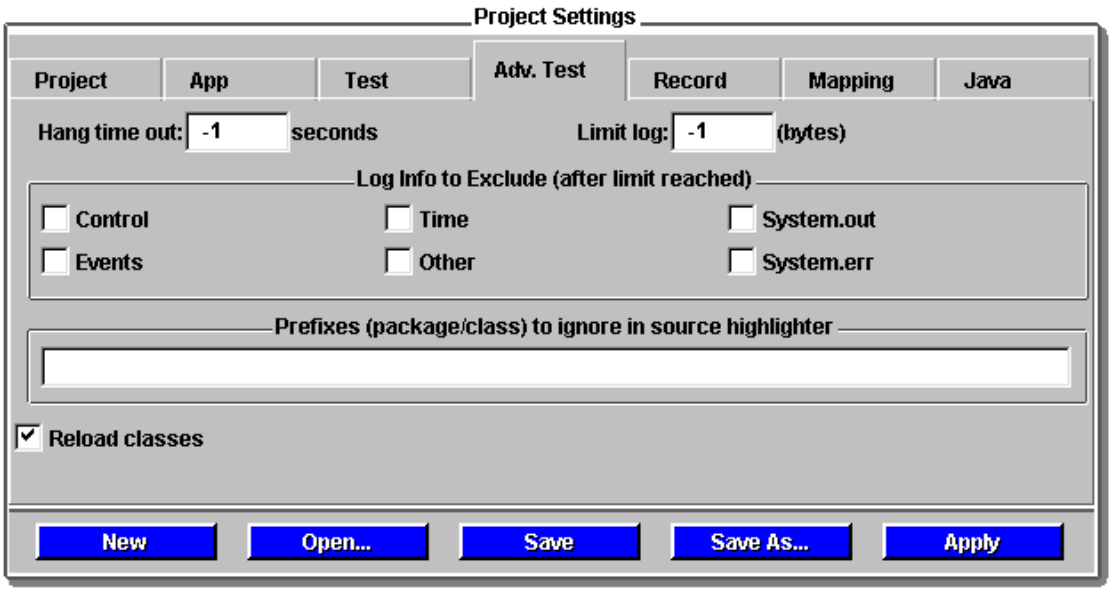

Figure 3-9 Adv. Test Panel

Advanced test options:

Hang time out	The number of seconds you want JavaStar to wait before determining that an unresponsive application is hung.
Limit log	The log file size at which JavaStar begins filtering.

Log filters	Select log file options from this list to specify which information you want to *exclude* from your log files. Unlike filtering you can do after a log file is generated (using command line options or the Results Viewer Extract option) setting Log limit filters means that the options you choose are never recorded to the log.
Reload Classes	Check this box if you want JavaStar to reload classes before executing a test. If you edit your tests while you have the Record / Playback window open, JavaStar will only reload the tests (and, thus, playback your changes) if you have this option turned on.
Prefixes (package/class) to ignore for source highlighter.	Packages or classes you want to define as libraries, so they won't be shown during playback. Instead, the call to the library will be highlighted.

Saving, Applying, and Loading Project Files

Use the buttons along the bottom of the Project Settings window to load, save, and apply Project Settings, as well as to clear fields in preparation for a new file.

Use...	**To...**
New	Clear the Project File settings and begin a new, unnamed file
Open	Load an existing project file
Save	Save the currently loaded project file using the same name
Save As...	Save the current project file under a different name
Apply	Apply the current Project Settings, so that these settings will take effect during this session

Recording Scripts 4

You record test scripts (`.java` files that JavaStar compiles) in the Record / Playback window. To get to this window, choose Create Script from the main menu, then enter the information to start your application or applet.

You can get step-by-step instructions for starting your program under test in the chapter "Preparing to Use JavaStar." See the section entitled Starting Your Application or Applet.

This chapter describes:

- Starting Record Mode
- General Recording Tips
- Comparing Values and Images within a Script
- Inserting Timers
- Editing Your Script While Recording
- Changing Options While Recording
- Pausing, Stopping, and Quitting

Starting Record Mode

To enter Record mode:

1. **If the Record/Playback window is not open, follow the steps in the** Starting Your Application or Applet **(the "**Preparing to Use JavaStar**" chapter).**

2. **If your application is up but the first dialog in the application is modal, use Ctrl-Alt-F7 to continue.**
   If Ctrl-Alt-F7 doesn't work on your system, try Ctrl-Shift-F7 or Ctrl-Meta-F7. The meta keys (when included on a keyboard) are marked with diamond symbols and located on either side of the spacebar.

3. **In the Record/Playback window, click** Record**.**
   Because JavaStar brings the application or applet to the forefront, you may need to click on the edge of the Record / Playback window to activate that window, or drag your program window to one side.

   The Record Test dialog window opens.

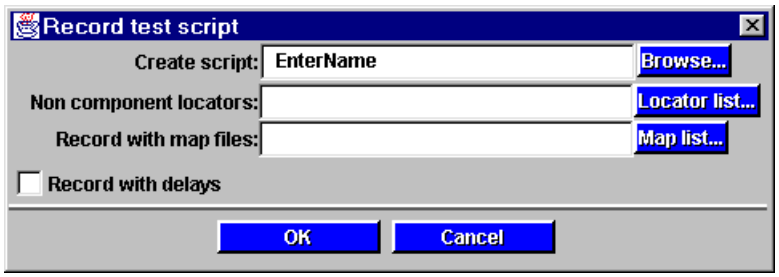

Figure 4-1 Record Test Script window

4. **Define the name you want to use for the script.**
   Type the filename directly into the Create script field, or browse to locate a test you want to overwrite. JavaStar creates a `.java` and `.class` file for the name you provide.

5. **If your program under test includes non-components, specify the locator you want to reference.**
   For a description of non-components, locators, and to learn how and when to use them, see the chapter "Locators for Non-Components." You need to read this chapter if you are testing an application or applet that uses a toolkit that does not derive from Java AWT.

   Note – JavaStar provides locators for the Bongo and IFC toolkits.

6. **If you want to record with declarations, enter the name of the declaration files you want to reference in** Record with map files.
   This feature is useful if you've already recorded declarations for your program under test and have edited those declaration files to use abstracted names. By specifying the declaration files you want to use, you set JavaStar to record using these files, instead of using default component names.

   To specify the declarations files you want to use, click Map list... to select from a list. If you can't locate declarations you know you recorded, check to see if you have compiled the declarations into `.class` files.

   Note – You can learn more about declaration files in the chapter "Generating and Using Declarations."

7. **If you want to record using delays between events, toggle the** Record with delays **checkbox on.**
   When you record with delays, JavaStar notes the length of any delay between recorded events and includes that delay in the test. Usually recording with delays is not necessary, but if your application has a Canvas component, or if it uses a non-AWT toolkit without a Locator, you need to toggle this option on. If delays aren't important to your test, leave this option off.

8. **Click OK.**
   The Record / Playback window opens (see Figure 4-2). The action buttons appear along the left side of the window.

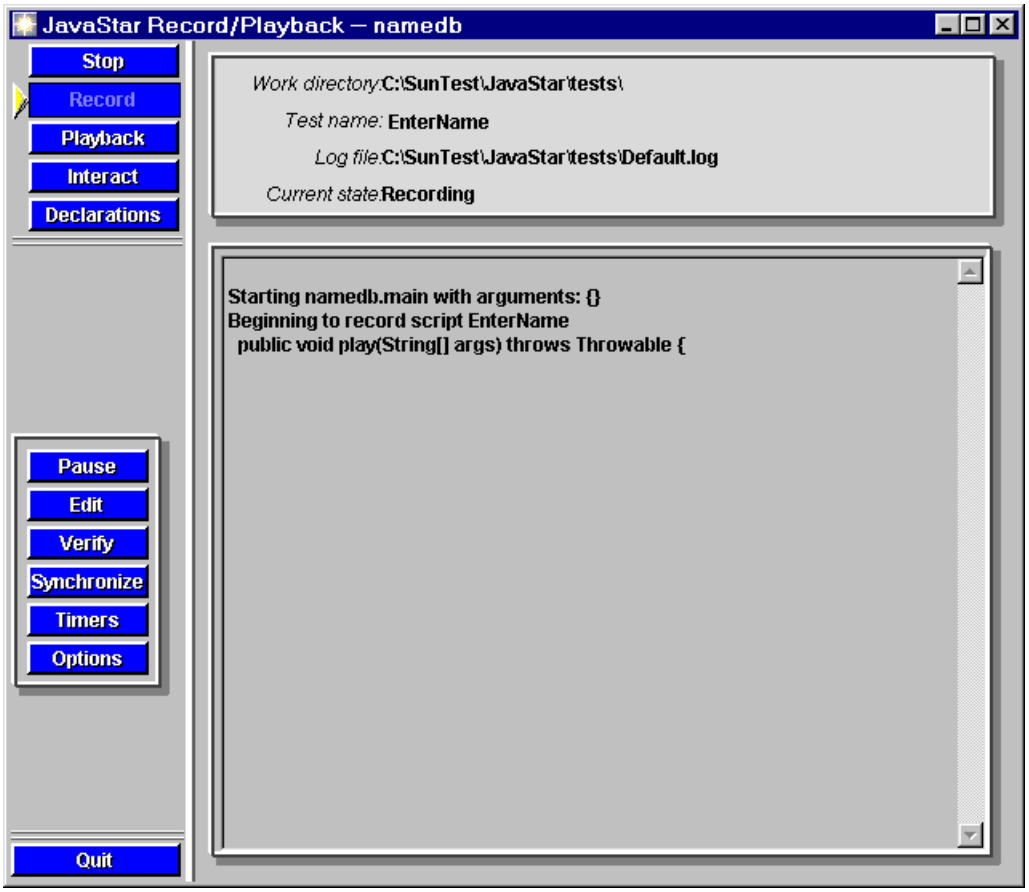

Figure 4-2 Record / Playback window during record mode

9. **Test your application or applet by interacting with the program directly.**
   JavaStar records your actions and dynamically displays the Java code it generates in the log panel.

10. **Click** Stop **to end recording.**

4

General Recording Tips

Recording keystrokes and mouse events is a straightforward procedure in JavaStar; once you've defined your test script name, you interact with the program you are testing exactly the way you want to test it. However, even given this simplicity, there are some important things you should know before you begin:

- **Make sure your record options match your needs.** The Record / Playback window uses options that you can set using the Set options button on the main menu, or by clicking the Options button that appears when you're in record mode. Record options specify which actions JavaStar records and which ones the application ignores. This can be useful if you want to reduce the size of your script and your log file. You can also set options for using regular expressions to find windows and dialogs, and to use component names internal to the program under test. For more details, see Changing Options While Recording or the chapter "Customizing Options."
- **Don't click** Stop **until you are sure you don't plan any more additions to the script.** Once you've stopped recording using the Stop button, you can't restart it again for that script—you can only edit the script manually. Use Pause if you want to suspend recording temporarily while you do something else. (Though, if you are not using delays while recording, leaving a test unattended for a period of time doesn't have any effect on the script.)
- **Don't quit or exit your application while recording**. The JavaStar record activity "lives" in the same address space as your program process. While this means JavaStar can access more information about your application or applet, it also means that quitting your application kills the recording process. Rather than shutting down the test application, use the restart nodes (in Compose Test) to end and restart a program. You can write a separate test that tests the program exit conditions.

Comparing Values and Images within a Script

JavaStar provides two types of comparison features: Verify and Synchronize. Both allow you to specify all or a portion of the window for comparison to recorded states. You can compare the attributes of a component or an image of what the component looks like. The difference between Verify and Synchronize is what happens during playback.

When you play back a test, a Verify operation performs the comparison as you specify. If the comparison fails, Verify attempts the comparison again, repeating attempts until it reaches the time-out value (which you can set in Playback Options). If the comparison fails at time-out, Verify notes the failure and the script continues to execute. The script itself does not fail and can end normally, though the log contains a failure that you can examine when you view the results.

A synchronize operation is the same as a verification,with one exception. When the timeout expires without a match, throws an `SyncException` and ends script execution, rather than logging a failure and continuing.

Choosing Between Verify and Synchronize

Use Verify when you are verifying attributes of a component (or an image) that are not critical to the continued operation of the script. For example, if you are checking the contents of a text field where the results won't impact the ability of the script to continue, Verify is a good choice.

Use Synchronize when your script needs to wait for a particular state change in the application before it can progress, or when the results of a comparison are critical to the script being able to finish. For example, if you expect the application to enable a button that you want to validate, and if you require that button for further testing, it makes sense to choose Synchronize over Verify.

> ***Note –*** The JavaStar API provides a third way for you to do comparisons—the `JS.check()` function. This method does a boolean check of an attribute or value you specify. You use `JS.check()` by editing your script to include the call; JavaStar does not insert these checks automatically while recording. For information on `JS.check()`, see check(boolean, String) in the "Component and Control Classes" chapter of the *JavaStar API Reference*.

How JavaStar Compares Component Attributes

When you select a component for comparison, JavaStar compares the attributes of the component and the attributes of its superclass. JavaStar doesn't compare every attribute, just the most relevant ones. For example, when you compare the attributes of a button, JavaStar compares the label, but not the foreground and background color.

Table 4-1 lists the attributes that are compared for each component type.

Table 4-1 Comparison of Attributes

If you compare a...	Then the following attributes are compared...
Button	Label
	Action command
Canvas	Graphics (optional)
Checkbox	Label
	State
CheckboxMenuItem	State

Table 4-1 Comparison of Attributes

If you compare a...	Then the following attributes are compared...
Choice	All items Selected index Selected item
Component	Is it enabled? Is it being shown? Is it visible? What kind of cursor? Popup menus
Container	All components have to match. For BorderLayout, the components are compared according to their orientation, i.e., only east, west, north, south, and center are compared. The rest of the components are ignored. For layout other than BorderLayout, components are compared according to their position in the component array returned by the getComponents() method.
Dialog	Title Is it modal? Is it resizable?
FileDialog	Mode (load or save) Directory File
Frame	Title MenuBar Is it resizable?
Label	Alignment of text Text
List	Does it allow multiple selections? All items Number of visible lines Selected items
Menu	All MenuItems Is it a tear-off Menu?
MenuBar	All Menus HelpMenu

Table 4-1 Comparison of Attributes

If you compare a...	Then the following attributes are compared...
MenuItem	Is it enabled? Label Action command Short cut
Panel	Graphics (optional)
PopupMenu	No additional attributes
Scrollbar	Unit increment Block increment Minimum value Maximum value Orientation Current value Visible amount
ScrollPane	Scrollbar display policy Size of the view port Current scroll position
TextArea	Number of columns Number of rows Visibility of scrollbars
TextComponent	Text Is it editable?
TextField	Is echo char set? Number of columns Echo char
Window	Warning string

Selecting Components

When setting up a comparison, use the mouse and keystrokes described in Table 4-2. When you select a component, the component flashes to show you what you selected, and the selection code appears in the Select for Verification or Select for Synchronization dialog.

Once you select a component, you can use the arrow keys to further navigate. The up arrow selects the parent of the component and the down arrow selects its first child. The right and left arrows move to the components that precede and follow this component, respectively.

Table 4-2 Selecting GUI Components for Comparison

If you want to select...	Then...
A frame or dialog window	Click a component of the frame and press F1.
Any other kind of component	Click it.

> ***Note –*** You can't select a menu or menubar for verification or synchronization. You can only verify these components through the containing frame.

Selecting Data Members and Methods

The Verify and Synchronize functions let you compare return values for one or more of a component's data members and simple methods. To do this, you select the option to compare using simple methods and data members. The procedure described in the section Verifying or Synchronizing Components explains the steps in detail. What's important to know now is how JavaStar defines the terms.

A *simple* method is any method that does not take any parameters and that returns a basic Java type—for example, int, long, float, and boolean, as well as `String`, `java.awt.Point`, `java.awt.Color`, `java.awt.Rectangle`, and `java.awt.Dimension`.

Note that if you query for the return value of a method while you're in the Verify or Synchronize screens, JavaStar executes the method. If the method you choose alters the state of the component, this could create a false comparison, so use caution.

Gold Files and Directories

A gold file is where JavaStar records the attributes of a component for comparison purposes. You generate gold files when you use Verify or Synchronize to compare using gold file attributes. Because the component you select might be a frame, gold files also contain the attributes for child components, as well. When you compare failure results later on, JavaStar reconstructs the components using these attributes. If an attribute includes an image (such as an image on a Canvas) that image is also saved to the gold file.

JavaStar stores gold files in a directory named *scriptname*`.gold`, located in the work directory. All comparison images are named sequentially, using the format `C1`, `C2`, `C3`, and so on. When you later run a test, any gold file comparison failures are stored in the failure directory you define. Failure files use the name format of `f.` followed by the gold file name. So, if the `C2`

comparison fails, you'll find the `C2.f` file in your failure directory, showing the component attributes during the test run. The gold files are not changed until you specifically specify that you want them replaced or deleted.

Other chapters that discuss gold files:

- Browsing Gold Files the chapter "Editing Tests" describes how to use the Script Editor to view gold files referenced by a specific script.
- Viewing Comparison Failures and Updating Gold Files in the chapter "Viewing and Analyzing Results," explains gold file management.

Caution – While you can open a gold file within a text editor, do not edit these files by hand. This could challenge the integrity of your test. Instead, use the Gold File Manager described in "Viewing and Analyzing Results."

Verifying or Synchronizing Components

The procedure and the screens for verifying and synchronizing are identical, except for their titles. Because the steps to use these features are identical as well, this procedure shows screen shots just one of the two options (Verify).

Remember that even though the comparison procedures are the same, verifications and synchronizations do not cause the same types of failures when the comparisons don't match. A verification logs the failure and continues, while a synchronization ends script execution with an exception. This is the only difference between the two comparison types.

This procedure assumes you are already in record mode and have progressed your test to the point at which you're ready to do your comparison.

1. **Click** Verify **or** Synchronize**.**
   This pauses recording and changes the right side of the Record / Playback window to show the Verify or Synchronize screen.

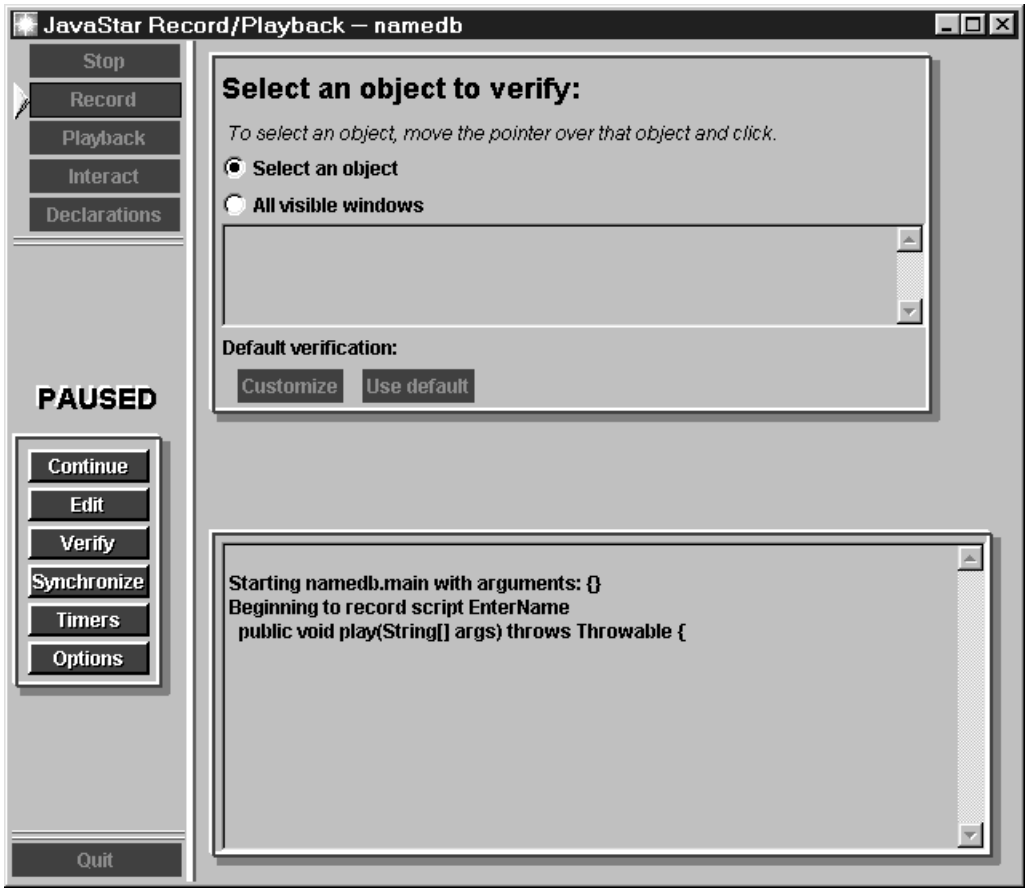

Figure 4-3 Verify—first screen

2. **Choose whether you want to compare a single component or all visible windows of your program under test.**
   Select an Object is the default option. If you choose All visible windows, Verify ignores any components you select next.

3. **If you chose** Select an Object**, move the mouse to your program under test and click the component you want to verify.**
   You can select any GUI component in your application or applet. Table 4-2 describes how to select the different types of GUI components.

   Once you click on a component, the selection code appears in the text portion of the verify screen.

4. **Decide whether you want to use the default comparison JavaStar suggests or if you want to customize the comparison.**
   Once you select All visible windows or click on a component, JavaStar updates the screen to present you with the most likely comparison you might want to do. For example, if you chose a button, JavaStar suggests a comparison *using enabled*—which means to compare and see if this button is enabled. Refer to Figure 4-4 to see this example.

   To accept the default, click Use default, then skip ahead to step 8.

   To change the default, click Customize.

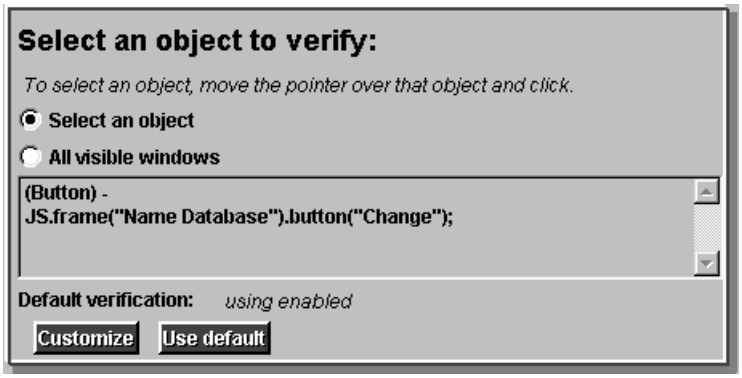

Figure 4-4 Verify—first screen with a component selected

5. **Choose the method of comparison you want to use.**
   The screen now shows your options for comparison (see Figure 4-5.)

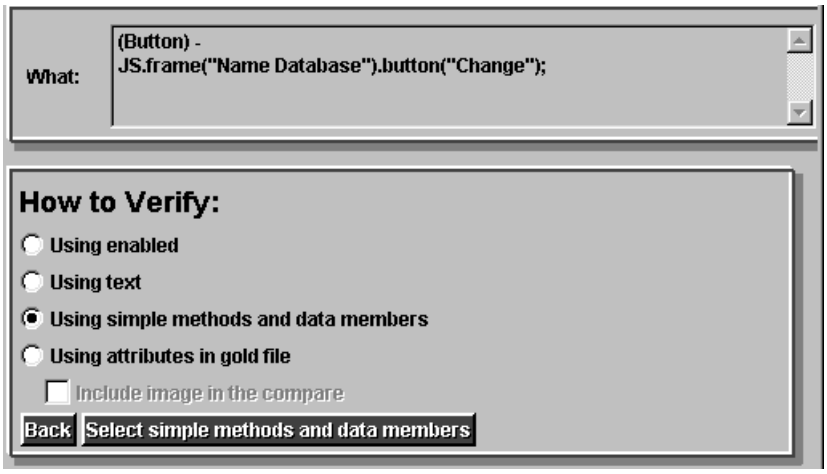

Figure 4-5 How to Verify screen

Choose your comparison type based on these descriptions:

Table 4-3 Comparison Options

Verify…	**Means…**
Using enabled	Verify that the component selected (such as a button or a panel) is enabled.
Using text	Compare the text of the selected component. The definition of component text depends on the type of component. For a text entry field, this is the text inside the field. For a Button, it's the Button Label. For a Frame, it's the title, and for a Label, it's the text of the label.
Using simple methods and data members	Compare the return value for this component's data members or methods. You'll be able to select which data members and methods you want to compare when you advance to the next screen.
Using attributes in gold file	Compare the component text and attributes to the information stored in a gold file. The gold file is generated when you set up the comparison.
Include image in compare	When comparing the contents and attributes of the selected component, also perform an image comparison. This option is only enabled when the component you are comparing is a canvas or a panel containing images. You can only select this option once you've turned on Using attributes in gold file.

6. **If you chose to compare using a method other than simple methods and data members, skip to the next step. Otherwise, click the** Select simple methods and data members **button.**
   For simple methods and data members, the screen changes to present you with a list of the data members and methods (including inherited ones) available for this component. See Figure 4-6 for an example. The list is presented alphabetically by field / method name.

   At any point, you can examine the current return value for a method by clicking the returns button, or get the value of a data member by clicking the = button. This does not select the item for comparison—it just tells you what the item returns.

   Note – Use caution when clicking returns, as this executes the method. If the method alters the state of the component, this could create a false comparison.

   To select one or more items for comparison:

a. **Scroll to the item you want to compare.**

b. **To highlight the line, click anywhere on the line other than on the** returns **button.**

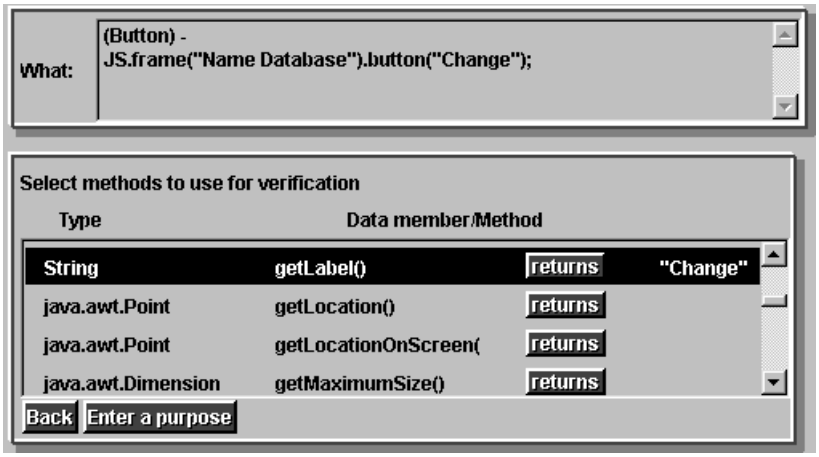

Figure 4-6 Verify—data member / method selection

7. **Click** Enter a Purpose**.**
   The screen changes to show you the component you have selected (What), the type of comparison you chose (How), and leaves a blank for you to provide a purpose (Why).

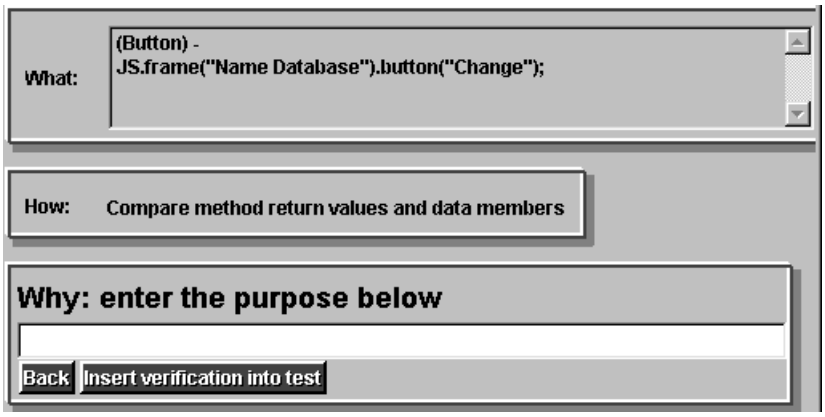

Figure 4-7 Verify—enter a purpose

8. **Type a purpose for the comparison in the** Why **field.**
   The string you enter shows up in the log file, and can be helpful when evaluating failures.

9. **Click** Insert verification into text **to record the comparison.**
   JavaStar records the attribute values or gold file to compare and adds the comparison code to the script. You can see the comparison code in the Record / Playback window. After you click Finish, the Select for Verification / Select for Synchronization dialog stays open, so you can define more comparisons.

10. **When you are done defining comparisons, click the** Continue **button in the left panel of the window.**
    This closes the Verify / Synchronize screen and returns you to record mode.

Inserting Timers

You can insert timers into your test that start and stop as you specify. When you run the test, the time values for each timer is written into the results log file.

- Timers can span multiple scripts. In this case, you define the start point in one script, define the stop point in another, and use these scripts together in a JST.
- Any timers for which you have not specified a stop point are not valid timers and will not show up in the results log.
- Timers display results in seconds. A time less than one second appears as `-1` in the log file.

To add a timer to your test:

1. **While in record mode, click the** Timers **button when you reach a point in your test where you want to begin a timer.**
   This pauses recording and opens the timer screen in the right side of the Record / Playback window.

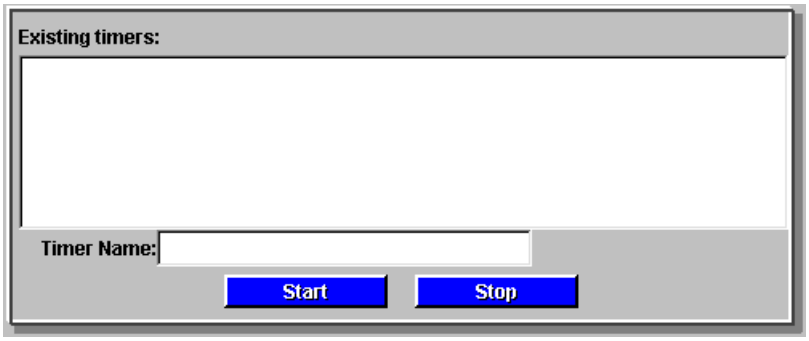

Figure 4-8 Timer screen

2. **Type a descriptive name for the timer in the** Timer name **field.**

3. **To set the start point, click** Start.
   JavaStar inserts a line of code into your script to start the timer.

4. **Click** Continue **to resume recording**.

5. **If you want to stop the timer in this script, click the** Timers **button when you reach a point in your test where you want to insert the timer stop point.**
   Record mode pauses and the timers screen opens again.

6. **Double-click on the name of the timer you want to terminate and click** Stop**.**
   Make sure that the timer name shows up in the Timer Name field—it won't if you single-click. Alternatively, you can type the name of the timer. Once you click Stop, JavaStar inserts the timer stop command into your script, and the timer is removed from the list of Existing Timers.

7. **Click** Continue **to resume record mode.**

Editing Your Script While Recording

While recording a script, you can pause to edit the script, inserting Java code or changing statements as necessary. Using the insert references option, you can select a component in your program under test and have JavaStar insert the reference automatically, speeding up your code additions and improving accuracy. When you're done, you can compile and save your changes, then continue on recording events as before.

The Script Editor that JavaStar opens in the Record / Playback window is not the same as the window you get when you click Edit Test Script in the JavaStar main menu. The Record / Playback editor includes a subset of the main Script Editor features. It also includes the Insert Reference feature, one specific to editing in record mode.

This section describes how to enter the script editor from record mode and how to use the Insert Reference feature. For information about the common features of both editors—Class Browser, Find/Replace, and Go to Line—see the chapter "Editing Tests."

Entering Edit Mode

1. **In the Record/Playback window, click the** Edit **button.**
   JavaStar automatically pauses recording and opens a script editor inside the right panel of the window. See Figure 4-9. You can now edit the code, adding or changing lines, as you want.

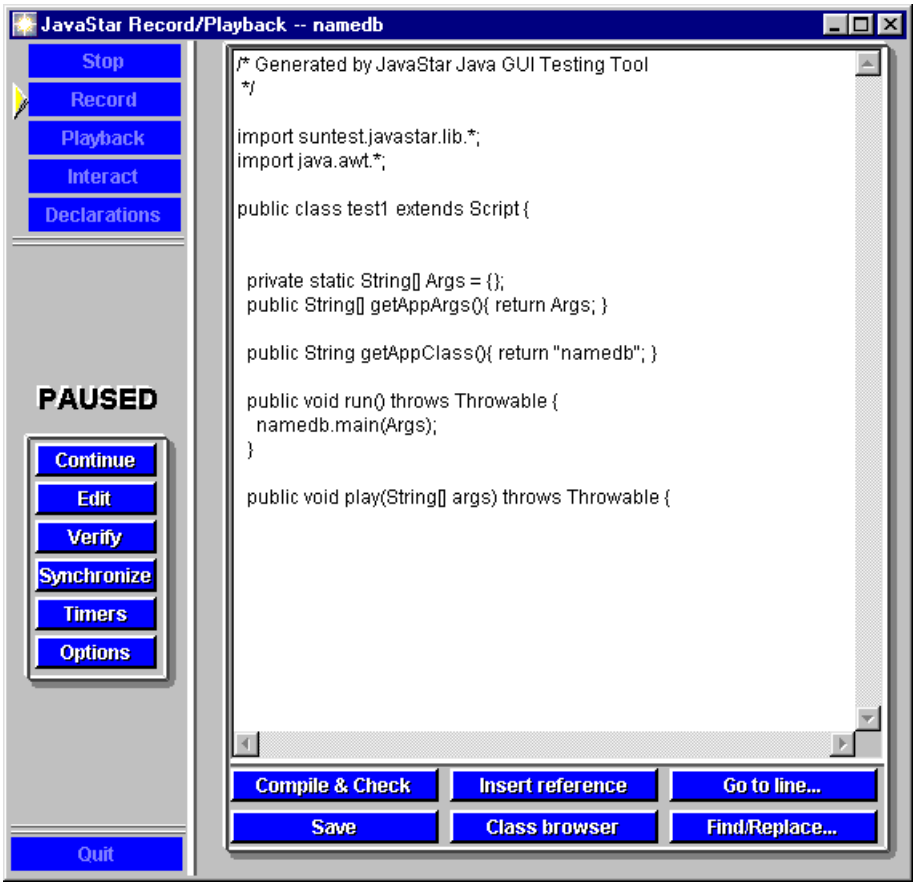

Figure 4-9 Script editor in the Record / Playback window

2. **When you complete your changes, click** Save**.**
   You don't need to compile your script because JavaStar will compile and check the code itself when you stop recording.

3. **Click the** Continue **button in the left panel to resume recording.**
   Now you can continue recording events as before, with your code changes incorporated into the script.

Inserting a Reference into a Script

With Insert Reference, you select a component and provide a variable name, and JavaStar then inserts a properly casted declaration, assigned to the variable name you specified. This is useful when you want to write your own verification code for more complex methods, such as methods that require parameters, or that return non-basic Java values.

To insert a declaration into a script you are recording:

1. **Click the** Insert reference **button.**
   The Insert Reference dialog is displayed. See Figure 4-10.

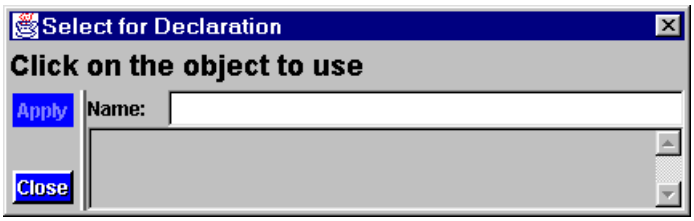

Figure 4-10 Insert Reference dialog

2. **In your program under test, click on the component you want to use.**
   The declaration for the component appears in the lower text box of the Select for Declaration dialog.

3. **In the** Name **field, enter the name you want to use in your code to identify this component.**
   Figure 4-11 shows a Select for Declarations dialog with a component selected and a name provided.

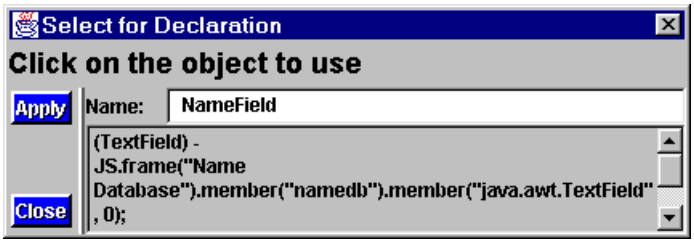

Figure 4-11 Insert Reference dialog with fields filled in

4. **Click** Apply **to insert the object reference into your script.**
   You can now, in subsequent code, refer to the component solely by the name you defined.

Changing Options While Recording

The Record / Playback window provides a shortcut to changing the record options during a recording session. This is useful if you've started recording and realize that the script is recording events you don't want to see in the log—or *not* recording events critical to your test.

This procedure assumes you are already in record mode.

1. **In the left panel of the Record/Playback window, click the** Options **button.** The recording options are displayed in the right portion of the window. See Figure 4-12.

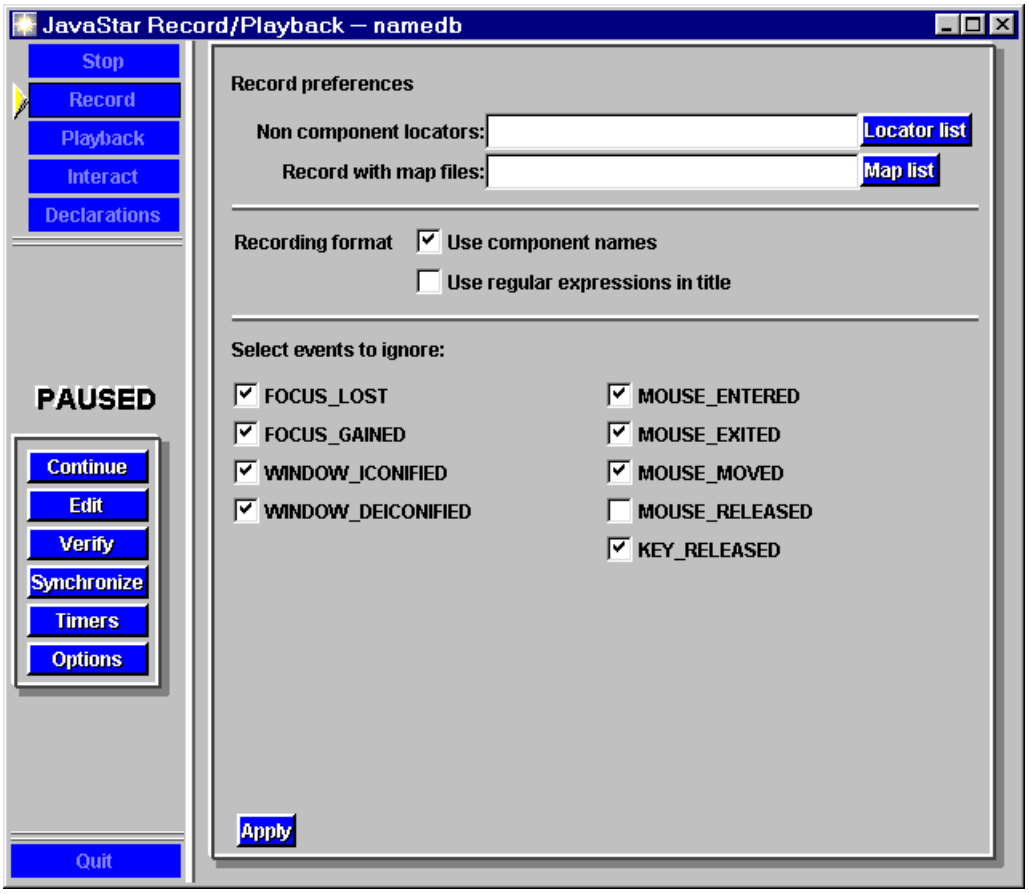

Figure 4-12 Recording options in the Record / Playback window

2. **Change the options according to your preferences.**
   For a detailed explanation of each recording option, see the Recording Format Options section of the chapter "Creating Project Files."
3. **Click the** Apply **button in the options panel.**
4. **Click the** Continue **button to resume recording.**

Pausing, Stopping, and Quitting

The Record / Playback window has Pause, Stop, and Quit buttons. They operate somewhat similarly—suspending or terminating a process—but they are not interchangeable:

- Pause suspends recording or playback so that you can take a break without affecting the process, or access options only available when the process is paused.
- Stop terminates the current playback, record, or generate declarations process. JavaStar ends the script and compiles it. Thus, once recording is terminated, you cannot start from the point where you left off—you need to manually edit the script to add to the test.
- Quit terminates any current playback, record, or declarations process and closes both the program under test and the Record / Playback window.

4

Interacting and Inspecting 5

To work with your application or applet without recording your actions, you can use the Interact feature of the Playback / Record window.

Interact is useful in several cases. Sometimes you need to interact with an application before beginning a recording session, particularly if you need to bring the program into a specific state. For example, you might need to load test data or clear fields of previous entries—tasks that you can take care of automatically within a composed test. For now, you can go into Interact mode, bring your test program into the proper state, then transition to Record mode and capture your interaction from that point on.

Interact is also helpful when you want to inspect a component of your test program to determine its state. Once in Interact mode, you'll see an Inspect button in the lower portion of the left button panel. By clicking this button, you can bring up a dialog window that lets you select a component, then query any simple data member or method of that component for its return value.

> ***Note –*** If you click the Interact button while recording, JavaStar stops recording and enters Interact mode. Selecting Interact while in playback mode terminates playback immediately.

Interacting without Recording

Interact is available only when no script is recording or playing back.

1. **Click** Interact**.**
   The Record / Playback window changes so that only the Stop and Quit buttons, as well as the new Inspect button, are active. You can now use your application or applet without JavaStar recording your actions to a script.

2. **Click** Stop **on the Record/Playback window to end interaction.**
   Interaction with the program under test is now disabled, and the Record / Playback window returns to its stopped state.

5

Inspecting Components

To inspect the return values for any variable or method of a component:

1. **If the Current State shown in the Record/Playback window is not** *Stopped***, click the** Stop **button.**

1. **Click** Interact**.**
   An Inspect button appears in the lower portion of the button panel.

2. **Click** Inspect**.**
   This brings up the Select for Inspection screen in the right portion of the window.

3. **Click on the component within your application or applet that you want to inspect.**
   You can navigate through components within a frame using the arrow keys. The up arrow moves you from a child component to its parent, while the left and right arrows move between sibling components.

   The text area of the Select an object to inspect panel shows the declaration for component you selected (see Figure 5-1).

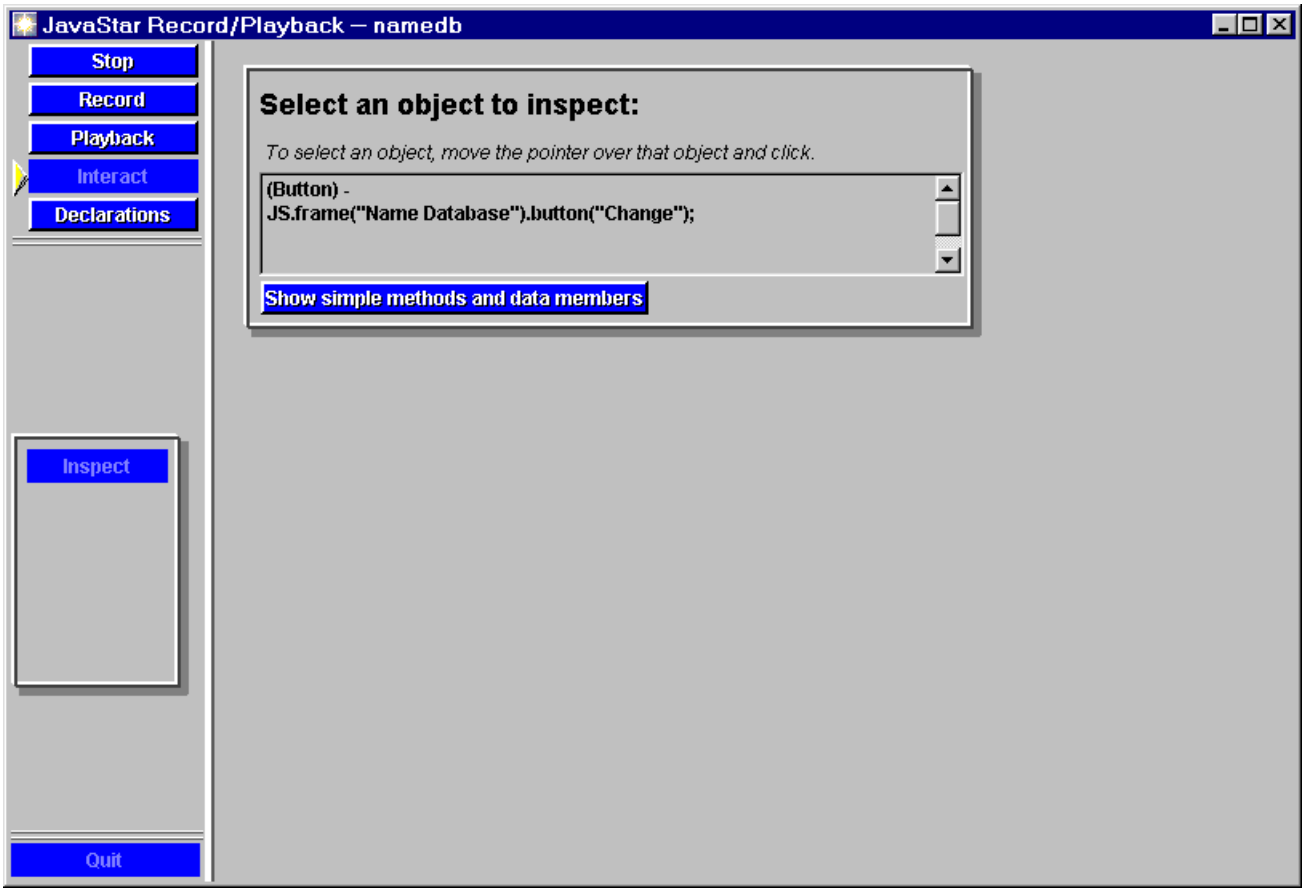

Figure 5-1 Select for Inspection—first screen

4. **Click** Show simple methods and data members**.**
   The screen changes to show a list of the simple methods and data members (including inherited members) for the component. The list is sorted alphabetically by method and member name.

5. **Navigate to the field or method you want to inspect and click the** returns **button for that item.**
   The return value is displayed to the right of the button. See Figure 5-2.

   Note – You need to rely on your judgement when querying a method for its return value. When you click returns, JavaStar executes the method you selected. If the method has a side effect, this may cause a problem.

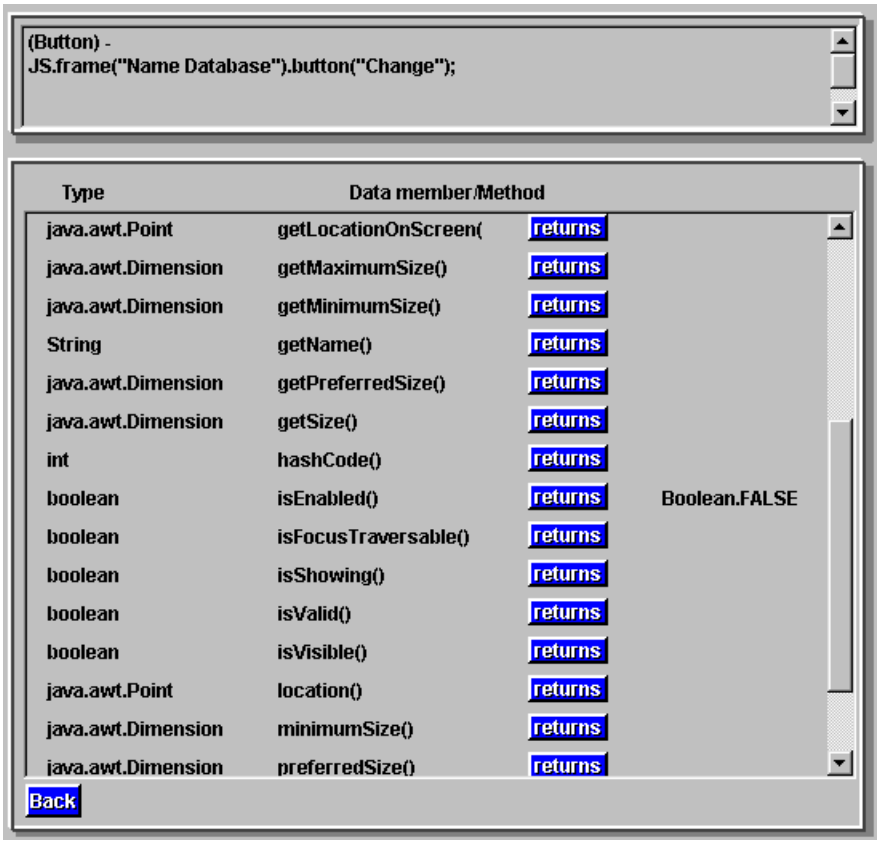

Figure 5-2 Select for Inspection—second screen

6. **To select another component, click** Previous **or** Reset**.**
   You can now repeat Step 3 through Step 5.

7. **To close the Select for Inspection window, click** Close **or** Finish**.**
   This keeps you in Interact mode but closes the window. You can click Stop to leave Interact mode.

Generating and Using Declarations 6

One of the problems with testing a GUI is that during program development the labels or positions of components often change, causing any tests on those components to fail. Another problem is that the default names can be so cryptic that reading source code and log files is unnecessarily slow.

JavaStar addresses these problems by providing an easy way for you to generate component declarations for each window in your GUI. These are written out as `.java` files, one file per window. Once you've generated these files (called *JavaStar declaration files*) you can then edit them to replace developer or default component names with more meaningful names. Then you compile these into classes and record scripts that reference these declarations files.

Now, when a component name changes you only have to edit its references in the JavaStar declaration file and recompile. All tests that reference the component will continue to run. Without declaration files, you need to update every script that references the changed component.

This chapter describes:

> ***Note –*** One situation declarations can't account for is when a developer changes the program interface such that the component now requires a different event type (which is typically the case when the component type changes). This problem is best addressed by using modular scripts that build into tests, where you only need to change the script that tests that component.

6

Generating Component Declarations

1. **Use** Create Script **to start the test program and launch the Record/Playback window.**
   Refer to the section Starting Your Application or Applet in the chapter "Preparing to Use JavaStar" for instructions.

2. **In the Record/Playback window, click the** Declarations **button.**
   An instruction dialog opens. Note that the Record / Playback window displays "Generate Component Declarations" in the status area, and the test program now accepts interaction.

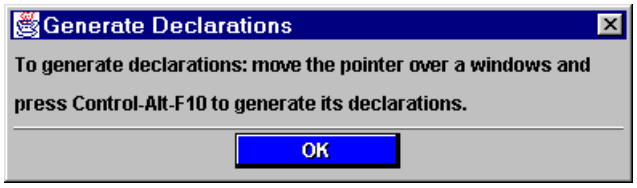

Figure 6-1 Generate declarations instruction dialog

3. **Move your pointer over a component (usually a frame), then press** Ctrl-Alt-F10**, or** Ctrl-Shift-F10**, to generate declarations.**
   A Generate Declarations dialog (see Figure 6-2) prompts you for the name of the package and class where JavaStar will write the declarations.

After moving your pointer over a window, you can use Ctrl-Alt-F10, Ctrl-Shift-F10, or Ctrl-Meta-F10 to generate declarations. If you are using a UNIX platform and Ctrl-Alt-F10 does not work, try Ctrl-Shift-F10. You can also use Ctrl-Meta-F10. The Meta key on UNIX keyboards is next to the spacebar, marked with a diamond symbol.

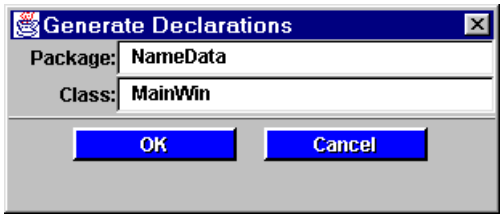

Figure 6-2 Generate Declarations

4. **If you want JavaStar to create a package for your classes, type the name into the** Package **field.**
   If you supply a package name, JavaStar creates a directory with this name within the current work directory. If you're generating declarations for all the windows of your test program, storing them in a package can make it easier to maintain and reference the declarations for an entire test suite.

   If you do not supply a package name, JavaStar saves all component declaration classes from this process into the work directory. JavaStar creates one `.java` file for each window you select.

   Note – If you do use a package name, you'll need to add the directory to your additional classpath, in recording and later in playback.

5. **Type the class name you want to use for this declarations file into the** Class **field.**
   JavaStar creates a `.java` file with the class name you enter. However, if the declarations file is loaded for the current session, the newly generated declarations files won't take effect immediately.

6. **Click** OK**.**

7. **To generate component declarations for other frames or dialogs in your application, bring them up and repeat steps** Step 3**—**Step 6**.**
   You can step through all frames of your test program and create component declaration classes for each one. Because the declarations for each frame are saved to a separate class, you can quickly detect changes and update portions of the component declarations at a later time.

8. **Click** Stop**.**

Editing Declarations Files to Use Abstract Names

One manual useful task involving declarations is the step of initially editing your declarations files to use abstracted names.

To do this:

1. **From the JavaStar main menu, select** Edit Test Script**.**

2. **Load the your first JavaStar declaration file into the JavaStar Script Editor.**

3. **Find the first declaration you want to abstract and edit this to use a name.**
   Lines with the `static` keyword contain the names.

   For example, if your file contains the code:

```
/* TextField */
public static JSComponent textField1(){
return Namedb().member("java.awt.TextField", 0);
}
```

and you want to change `textfield1` to a more meaningful name, replace it with the name you want to use.

Note – Be careful not to edit the declaration so it is unrecognizable to JavaStar. The signature of all methods in the declarations file has to be:

```
public static JSComponent name()
```

4. **Use the** Find/Replace **button to search the declarations for any additional instances of the declaration you changed, and update these too.**
5. **Repeat** Step 3 **through** Step 4 **for the remaining component declarations.**
6. **Click** Save & Compile **to create a `.class file`.**

   Note – Whether you edit a JavaStar declaration file or not, you need to save and compile declarations after generating them.

Using Declarations Files in Record Mode

If you specify your declarations files before you begin recording a test, JavaStar generates test code using your abstracted names.

1. **In the Record/Playback window, click** Record**.**
   This opens the Record Test dialog.
2. **In the** Record with Map Files **field, enter the list of declarations files or click the** Map list **button and locate them through navigation.**
   The names of map files (JavaStar declaration files) must be fully-qualified class names. For example, if you stored your declarations in a package called `NamedbMaps`, and your first declaration file is named `main`, type `NamedbMaps.main`. You need to list each JavaStar declaration file you want to reference. On Windows platforms, use a semi-colon as a separator; on UNIX platforms, use a colon.

   Note – The declarations files you want to us must already be added to the additional classpath.

   If you choose to use the Map list feature, you'll see that Javastar opens a multi-pane window.

   a. **Use the Select an Item pane (upper left) to navigate to your package to open it.**
      See Figure 6-3 for a screen shot of the Select Map classes window.

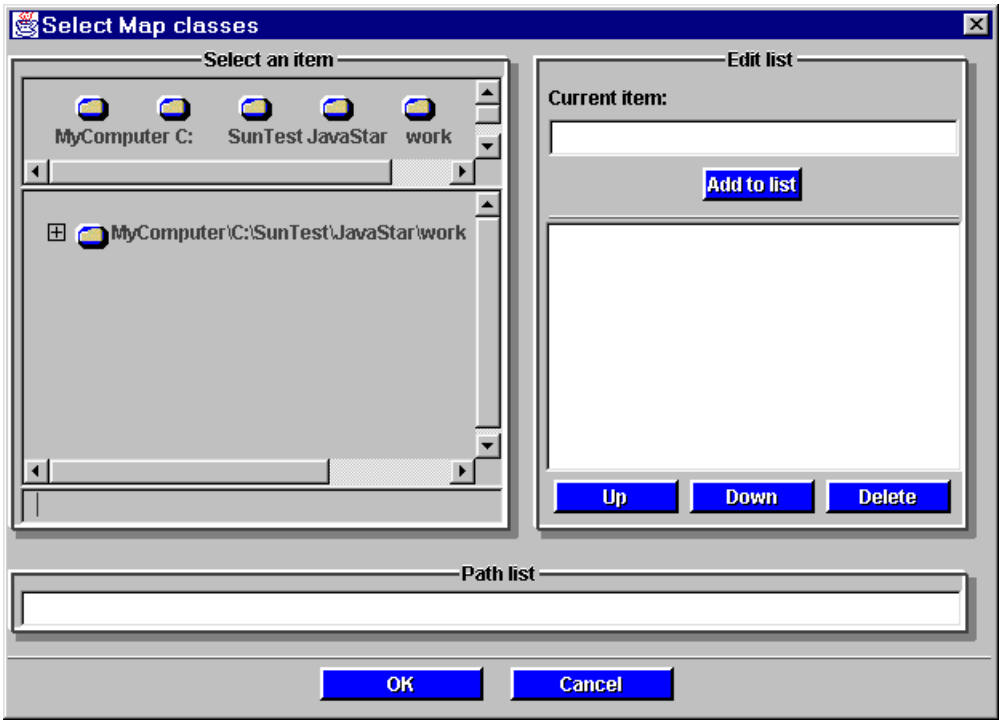

Figure 6-3 Select Map classes dialog window

b. Select a JavaStar declaration file to add.
This appears in the Edit list pane (upper right) in the Current items field.

c. Click Add to list **to move it to the list below.**
The file is also added to the Path list pane at the bottom. See Figure 6-4.

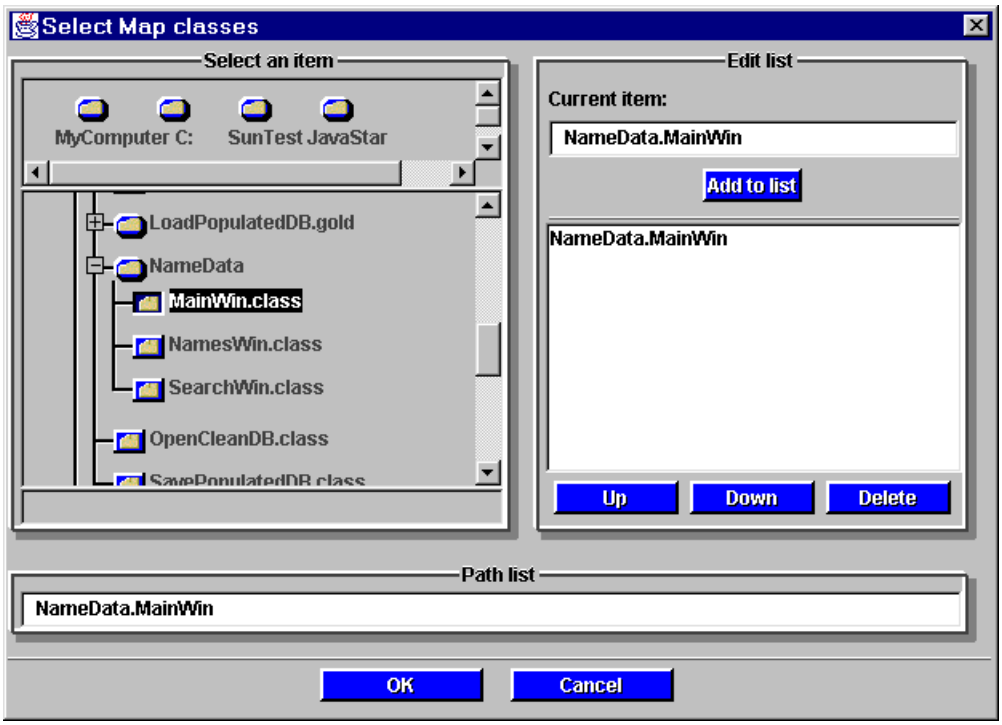

Figure 6-4 Select Map classes with list

d. Continue until you've added all declarations files.

e. Click OK**.**
JavaStar pastes the path list into the Record with Map Files field.

3. In the Record Test Script dialog, click OK **to begin recording.**

Modifying Existing Scripts to Use Abstracted Names

You are not restricted to incorporating declaration files by recording new scripts; you can also edit existing files to reference the declarations. Most often this is not the fastest way to make the change, but in the case of tests that include significant amounts of custom code, you might want to make the changes manually.

When you edit a script to use declarations, you need to:

1. Open the `.java` file for your script in the Script Editor.

2. Add a line of code to import the declarations.
Put this toward the top of the script, with other `import` statements.

```
import NameData.*;
```

3. **If you're using abstracted names, modify the component names within the script to use the new, assigned names.**
4. **Click** Save & Compile**.**

Composing Tests 7

Composing tests from scripts is where you start to access the real power of JavaStar. The Compose Test feature is a graphical interface that allows you to link scripts together to form a single test or group of tests. Each script becomes a *node* in the test tree you develop. These composed tests, called JSTs (JavaStar Tests) can be used to build more complex tests or groups of tests based on reusable test code modules.

Within a JST, you can:

- Handle recovery from test errors by setting a node to execute only if an exception occurs (for example, if a synchronize operation fails)
- Define that the application restart before executing a particular test node
- Specify constants to pass to scripts as arguments
- Define parameters to pass to nodes or scripts

While scripts are `.java` files that JavaStar compiles into `.class` files, JSTs are JavaStar-specific files that use a `.jst` extension. All `.jst` files are editable in the Compose Test window, and they can be incorporated as nodes in other JSTs to create even more powerful test suites. JST files do not contain java code; this is a JavaStar-specific format.

This section covers these Test Composer tasks:

7

Opening the Test Composer

To open the Test Composer:

- **Click the** Compose Test **button in the main menu panel.**
  The Test Composer window opens (see Figure 7-1).

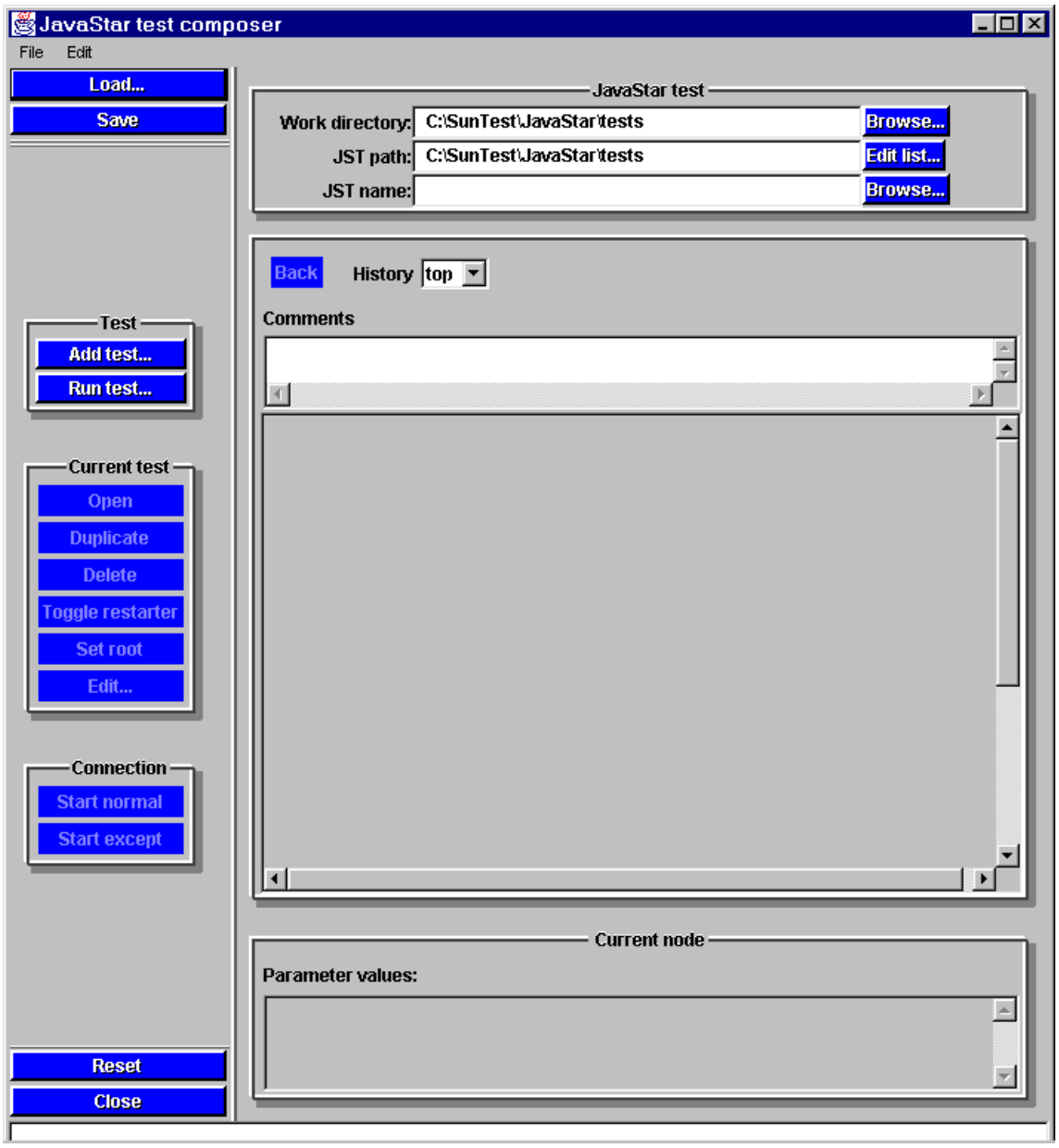

Figure 7-1 Test Composer

Setting the JST Path

The JST Path field is where you enter the directory paths you want JavaStar to search when looking for JST files. You can define multiple paths: for Windows, separate each path with a semi-colon; on UNIX platforms, use a colon. You can set the JST path in Project Settings, or you can enter it in the Test Composer.

To set the JST Path:

- **Type the paths into the** JST Path **field.**

If you want to use the same JST Path each time you run JavaStar, edit the JST Path value as described in the section Setting Test Options in the chapter "Creating Project Files."

Loading an Existing JST File

1. **Type the name of the JST in the** JST name **field or click** Browse **to locate the file by navigating through the file structure.**
   You do not need to type the `.jst` extension.
2. **Click** Load**.**
   JavaStar loads the test into the window. Figure 7-2 shows the Compose Test window with a JST loaded.

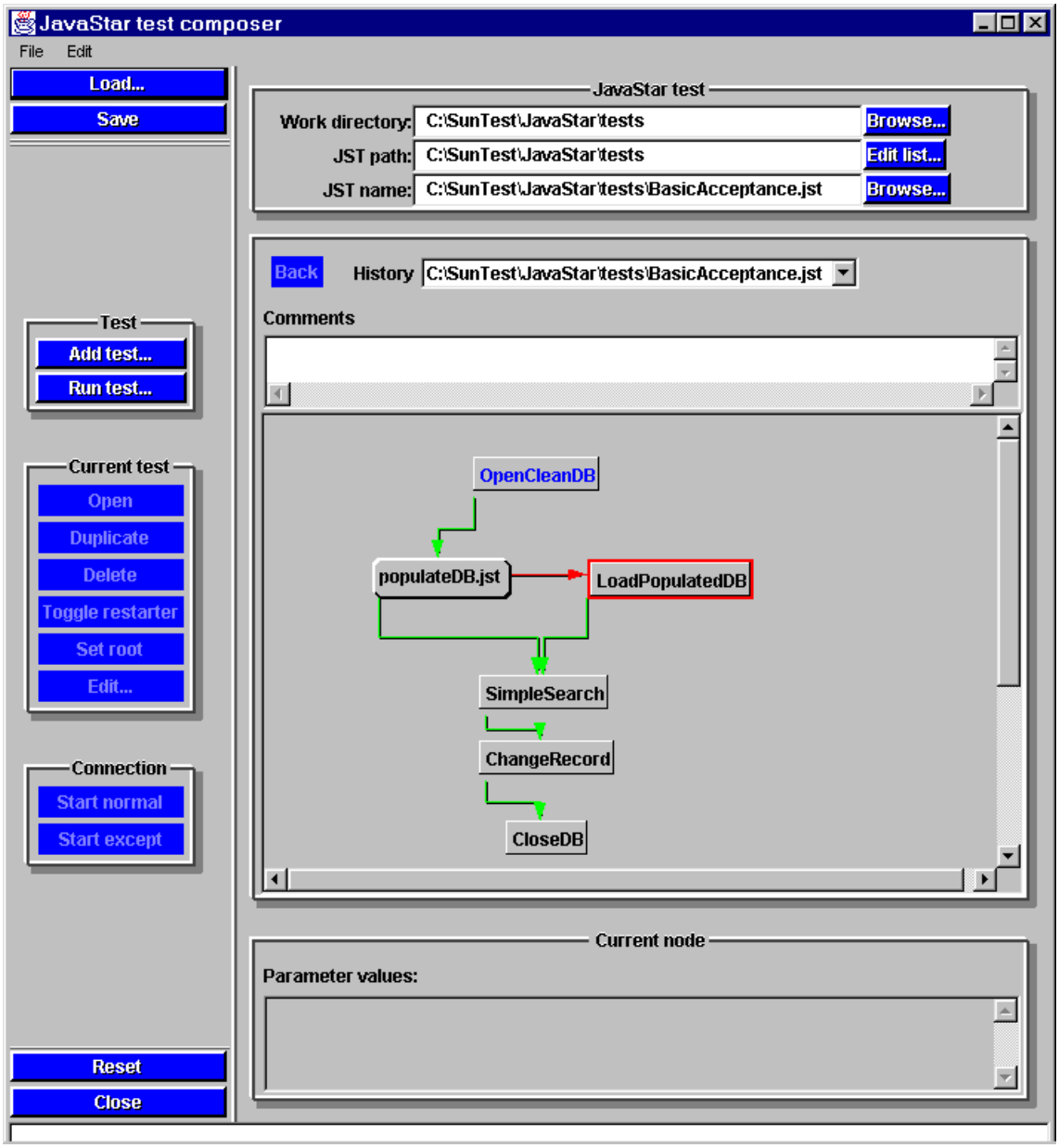

Figure 7-2 A JST loaded into the Compose Test window

Saving Tests

You must have at least one node in your test before you can save the `.jst` file. To save a file:

1. **Type the name of the JST in the** JST name **field.**
   You do not need to add the `.jst` extension.
2. **Click** Save**.**
   JavaStar saves the file to your current work directory.

Starting a New JST

If you just opened the Test Composer, you don't need to click Reset to start composing a new test. However, if you've been working with one or more JST files since you opened the Composer and want to start a new one, you can click Reset to clear the canvas and all field settings, including the JST name.

Composing a JST

When you compose a JST, you create nodes that correspond to test scripts or other JST files. Then you specify the dependencies between these nodes by setting normal and exception conditions. You can set any node to restart the application or applet under test before the JST executes the node. You can change the root (starting node) from the default. You can also specify parameters to send to a node, assuming the script it represents accepts parameters.

The basic rules about JSTs are:

- You can only have one starting point for the JST, and that is the root node. You can change which node functions as the root node, but you must have one. If you create a node not reachable from the main node, JavaStar will discard the node when you save the JST.
- You can include nodes for scripts you haven't yet written, but all JSTs you include within other JSTs must exist at the time you save them. JavaStar issues an error message if it cannot find a specified JST.

Most of the Test Composer functions are represented by buttons on the left side of the window (see Figure 7-3).

Figure 7-3 Compose test main buttons

You can also access these functions by putting your mouse cursor in the large open test area, holding down the right-mouse button, and selecting functions from the pop-up menu that appears. If you right click over an existing node or a connecting line, the menu displays only the options that apply to the node or connecting line.

This section describes:

- Creating a Node
- Running the Test
- Duplicating a Node
- Deleting a Node
- Setting a Node to Restart
- Choosing a Root Node
- Starting Normal and Exception Conditions
- Deleting a Connection
- Moving Nodes
- Adding Comments
- Editing a Node to Accept Arguments
- Editing Existing Parameters for a Node

Creating a Node

1. **Click the** Add Test **button or right-click in a blank area of the composer canvas.**
   The Test Name box appears, prompting you for the test you want this node to represent.

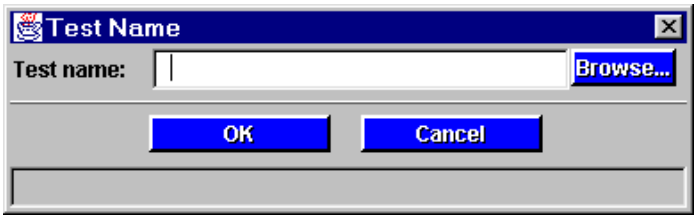

Figure 7-4 Test Name dialog

2. **Type the name of the script or JST you want to use, or click** Browse **to locate the file through the file dialog box.**
   JavaStar shows script nodes as rectangles with squared corners and JST nodes as rectangles with rounded corners. If the name you enter ends with `.jst`, JavaStar inserts a JST node into your test. If you enter a name without a `.jst` extension, JavaStar inserts a script node.

   A script does not have to exist for you to define it as a node. You can define nodes that represent scripts you plan to record or write at a later date. A JST, however, must exist for you to reference it as a test name.

3. **Click** OK**.**
   The node appears as a box with the script or JST title inside. If this is the first node you created for this test, it appears in blue to indicate that it is the root node.

Running the Test

The Run Test button is a short-cut to the Run Test command on the main menu. When you click this button, the Run Test dialog window appears, with your JST name already filled into the Test Name field.

You can now run the test as usual. Because you're running a JST, the JST Runner window appears with the Record / Playback window and your program under test. The JST Runner shows how JavaStar is progressing through the nodes (including nested JSTs) by flashing the currently executing node. If the test ends prematurely, you can see which node contains the problem, and edit it in the Test Composer or Script Editor, depending on the nature of the problem.

For information on how to use the Run Test window, see the chapter "Running Tests."

Duplicating a Node

Duplicating nodes is often useful, especially when you are creating a test that calls one script multiple times, changing only the parameters it sends with each call. All you have to do is duplicate and then edit the node.

1. **Click on the node you want to duplicate.**
   The node darkens to confirm your selection.
2. **Click** Duplicate**.**
   The node is duplicated, without links.

Deleting a Node

1. **Click on the node you want to delete.**
   The node darkens to confirm your selection.
2. **Click** Delete**.**
   The node is removed, along with any connection lines extending to or from the node.

   Note – You cannot delete a root node. If you want to delete a node currently set to root, first define another node as the root node (see Choosing a Root Node) then delete the original node.

Setting a Node to Restart

Restart nodes specify that when the JavaStar reaches this node during playback, the application under test will be restarted before executing the script (or JST) referred to by the node.

While you may have many reasons to restart a node during normal test functioning, the most common use is to handle exception conditions. By restarting the application, you return the test program to a predictable state, from which you can resume testing. Populating JSTs with exception conditions that link to restart nodes makes it easier to create long tests that run unattended—perhaps overnight—with reasonable assurance that the tests will complete, even with failures.

To set a node to restart:

1. **Click on the node you want to turn into a restart node.**
   The node darkens to confirm your selection.
2. **Click** Toggle Restarter**.**
   A red box appears around the node to signify that it is now a restart node.

To remove the restart condition from a node, you repeat the same procedure:

1. **Click on the node you want to return to a non-restart status.**
   The node darkens to confirm your selection.

2. **Click** Toggle Restarter**.**
   The red box around the node is removed, indicating that it is no longer a restart node.

Choosing a Root Node

A root node is the starting node for the test. You must have a root node for your test, and you can only define one node as the root. By default, the first node you create for a test is a root node. If you want to change the root node:

1. **Click on the node you want to turn into the root node.**
   The node darkens to confirm your selection.

2. **Click** Set root**.**
   A blue box appears around the node, indicating it is the root node.

Starting Normal and Exception Conditions

You link nodes together using normal and exception connecting lines, represented by green (normal) and red (exception) arrows.

A normal connecting line between two nodes means that the second node will only be executed if the first node ends normally. A normal condition means that the previous test did not throw an exception. This does *not* mean that all comparisons passed, because while a synchronization failure throws a specific exception, verification failures do not throw exceptions (though details are included in the test run log file).

An exception connecting line means the second node will only be executed if the first node ends with an exception. An exception condition means the script was terminated prematurely when an operation threw an exception. Exception connectors are particularly useful for introducing error recover into your tests—in combination with Setting a Node to Restart, your tests become quite powerful.

You can only have one normal and one exception condition starting at each node. There is no limit, however, on the number of lines of one type that connect to a single node.

To create a condition line:

1. **Click the node from which you want the condition to originate.**
   When the test executes, JavaStar will evaluate the exit state of *this* node before proceeding to the next.
2. **Click either the** Start normal **or** Start except **button.**
   Note that your selected node starts to flash. That's to confirm which node you chose and to let you know you have an operation in progress.
3. **Click on the node you want to execute next.**
   An arrow appears extending from your start node to your end node.

Deleting a Connection

1. **Click on a connector line to select it.**
   The line flashes to confirm your selection.
2. **Click the** Delete **button.**
   JavaStar deletes the line.

Moving Nodes

- **To move a node, move the pointer over the node you want to move, hold the mouse button down, and drag the node to another location.**

Adding Comments

Use the comments field to explain the purpose of this JST and any important information other test developers might need to know when maintaining this test.

Editing a Node to Accept Arguments

In the Compose Test window, you can select any node and specify arguments you want to pass to it at execution time. JavaStar supports three types of arguments (also called parameters):

- **Parent Parameters**—an argument to inherit from this JST's parent node. If this JST has no parent, then these parameters must be passed as test arguments when you run the test.
- **Constants**—values you specify while editing the node itself.
- **Property Names**—arguments that will be read from a properties file.

This section describes the basics of how to specify parameters for a node within a JST. For a detailed description of how to use parameters to make your tests more flexible or how to edit scripts to support parameters, see the *JavaStar Tutorial*.

To edit a node:

1. **Click on an existing node.**
   The node darkens to confirm your selection.

2. **Click on the** Edit... **button.**
   The Edit node dialog appears with the test name filled in.

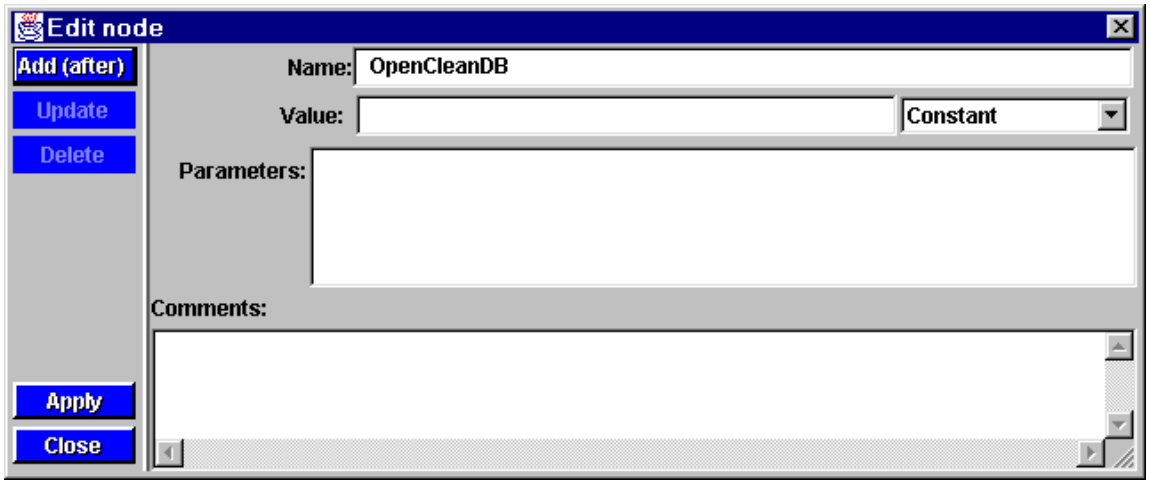

Figure 7-5 Edit Node dialog

3. **Select the type of parameter you want to pass to this node.**
   The pull-down menu to the right of the Value or Argument# field (set to Constant by default) provides three options. The label for the field to the left of the parameter type pull-down changes based on your choice—by default, the field label is Value.

Table 7-1 Parameter passing options

Parameter Type	Description
Parent parameter	Get this parameter from the parent node. If this node is part of a JST that does not have a parent, the parameter must be passed as a test argument at the time you run the test. Selecting this option changes the field name to the left to Argument#.
Constant	Pass the value as a constant. You specify the value for the constant in this window—note that the field to the left is labeled Value when you select this option.
Property name	Obtain this value from a Java property file. The field label to the left changes to Property Name when you select this type.

4. **Specify the value, argument #, or property name.**
   The field to the left of the pop-up menu prompts you for the type of information you need to provide for the selected parameter type.

   If you want to provide a constant, enter the value. For example, if you're providing a numeric value that your script references, type the number directly into the Value field.

   For parent parameters, refer to the parameter you want to use by argument number. Remember that argument numbers start at 0, not 1. For example, if three parameters are passed to the parent node, and you want to use the second parameter in this script, enter 1 as the argument number.

   Note – For property files, you need to specify the name of the property file in JavaStar Playback Options. You can only have one property file defined at one time.

   See Figure 7-6 for an example of an edited node.

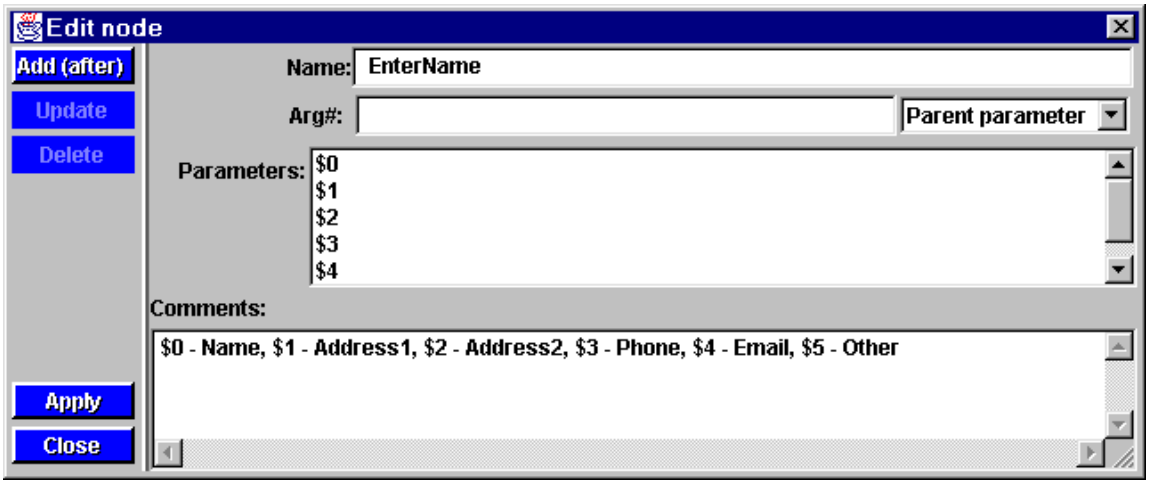

Figure 7-6 A commented node that defines five parent parameters.

5. **Click** Add (after) **to make this parameter the next in the list.**
   The parameter appears in the list box below.

6. **Continue adding parameters by repeating steps 3 and 4.**

7. **Add text to the** Comments **field to describe the parameters you are passing.**
   The comment field applies to the node itself, not a specific parameter. Adding a description here is particularly useful when you're passing parent parameters or using property names—you can save time you might spend later trying to decipher what you were trying to pass.

8. **Click** Apply **to put the changes into effect.**
9. **Click** Close **to close the window.**
   Note that now when you select this node, the parameters display in the Parameter Values text box, located at the bottom of the Test Composer window.

Editing Existing Parameters for a Node

With the edit node window open, you can delete a parameter, change the value, insert a parameter into the list.

To Delete a Parameter

1. **Select the parameter from the** Parameters **list.**
2. **Click the** Delete **button.**
3. **Click** Apply **to save the changes.**
4. **Close the window.**

To Change the Value of a Constant

1. **Select the constant from the list.**
2. **Edit the value in the** Value **field.**
3. **Click the** Update **button.**
4. **Click** Apply **to save the changes.**
5. **Close the window.**

To Insert a Parameter Between Two Others

1. **Click on the parameter you want to proceed the new one.**
2. **Enter the new parameter into the** Value **field.**
3. **Click the** Add (after) **button.**
4. **Click** Apply **to save the changes.**
5. **Close the window.**

7

Navigating Through Nested JSTs

If your JST contains a node that is also a JST (these are indicated visually by slightly rounded corners) you can use a single step to navigate to the contents of that JST. To do this:

1. **Click on the JST node in the Test Composer.**
   The node darkens to confirm your selection.

2. **Click the** Open **button.**
   The display changes to show the JST file for the node you selected.

To move up the tree:

- **Click the** Back **button or click on the** History **pop-up menu next to this button to choose the level you want to return to.**
  The Back button is located in the area above the JST comments field. Back always takes you back to the previous level. If you "drilled down" through several levels of JSTs, you can use the History pop-up menu to choose the level you want to return to.

Closing the Test Composer

To close the Test Composer window, click the Close button. JavaStar prompts you if you have not saved your latest changes.

Editing Tests 8

The Edit Script feature of the main menu allows you to edit scripts. Within this editor, you can view and use classes and methods of the JavaStar API library, and you can add your own custom code. You can also compile the code in the script editor. This is important because your code changes only take effect once it is compiled.

The changes you make to a script cannot be used by Record / Playback windows that were open at the time you made your edits. To use the script changes, you need to close the Record / Playback window and either choose Run Script or Create Script.

The JavaStar script editor is not intended to be a full-featured development environment. It is supplied for convenience, providing you with a quick way to make script edits, browse classes for available methods, and compile Java code. You can use any text editor or integrated development environment to make your changes. If you use a text editor, be sure to compile the scripts before running them.

This section describes:

8

Loading a Script to Edit

1. **From the main menu, click** Edit Test Script**.**
   The Script Editor window opens.

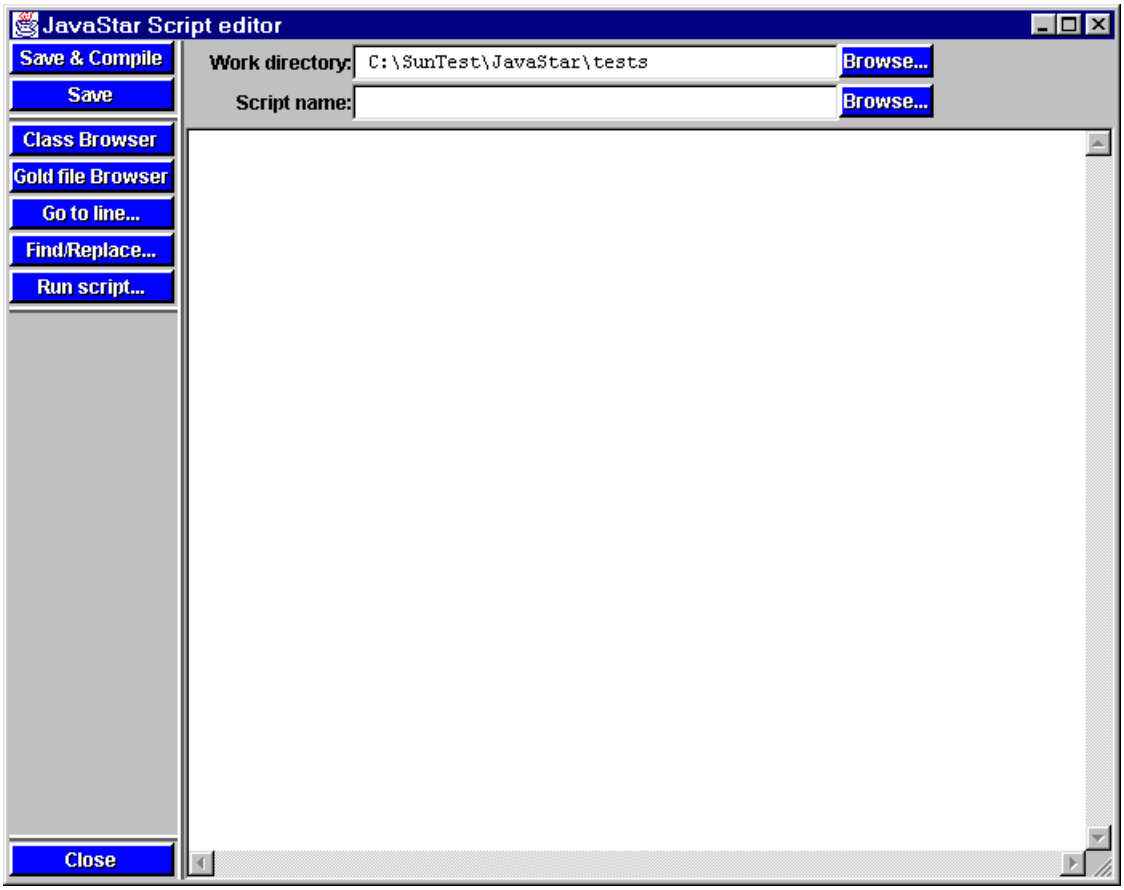

Figure 8-1 Script Editor

2. **Type the name of the `.java` file into the** Test name **field or browse to locate the file by navigating through the file structure.**
   Be sure to include the `.java` extension when typing the test name.

3. **Click** Load File **to load the file into the editor.**
   The code displays in the text edit window. You're now ready to edit.

Browsing Class Components

You can use the class browser to view the members of any Java class in the CLASSPATH, including the JavaStar libraries.

1. **In the Script Editor window, click** Class Browser**.**
   The Class Browser window opens.
2. **In the** Class **field, type in the full name of the class you want to browse and press** Return**.**

To load a JavaStar library:

1. **In the Script Editor window, click** Class Browser**.**
   The Class Browser window opens.
2. **Pull down the Library menu and select a library to review.**
   The library members are displayed in the text window.

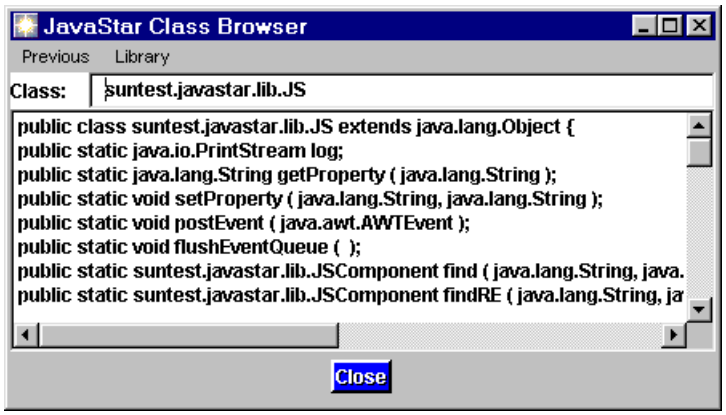

Figure 8-2 Class Browser

Browsing Gold Files

You can view the gold files for the current script from the Script Editor. To view the files:

1. **Within the Script Editor, open the script that references a gold file.**
2. **Click the** Gold file Browser **button in the left button panel.**
   The Gold file viewer dialog opens.

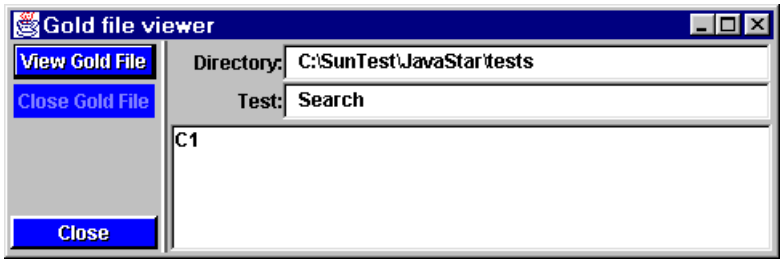

Figure 8-3 Gold file viewer

3. **Select the gold file you want to view.**
   Gold files are numbered sequentially starting with `C1`. If no filenames appear in this list, there are no gold files for this script.

4. **Click** View Gold File **to see the gold file as an image.**
   JavaStar displays a window with the image.

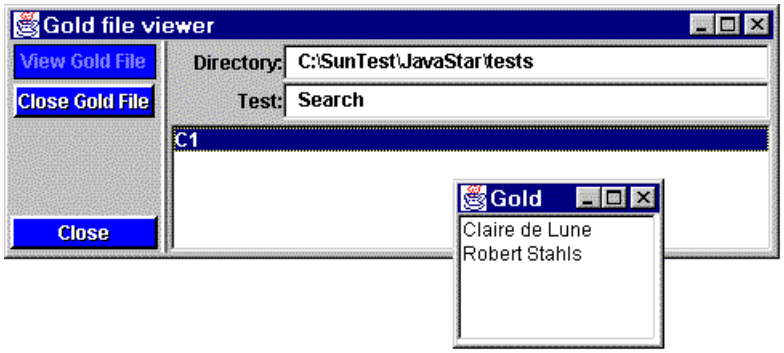

Figure 8-4 Gold file viewer and gold file

5. **Click** Close **when you are finished.**

   Note – If you want to update the contents of the gold file with new results from a test run, use the View Failure button in the Results Viewer to access the Gold File Manager. You can update the gold file from there.

Going to a Specific Line Number

1. **Click the** Go to line... **button**
   The Go to line number dialog appears.

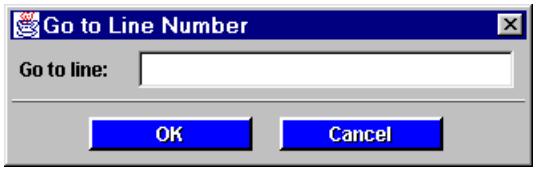

Figure 8-5 Go to Line Number dialog

2. **Type the number you want to go to.**
3. **Click** OK**.**
   The editor advances to the corresponding line in the script.

Finding and Replacing Text

1. **From the Script Editor window, click the** Find/Replace... **button.**
   The Find / Replace dialog opens.

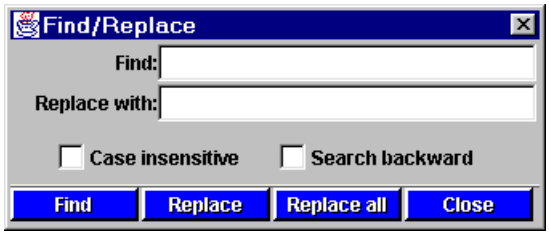

Figure 8-6 Find / Replace dialog

2. **Type the text you want to find in the** Find **text field.**
3. **To replace the text, type the replacement into the** Replace with **field.**
4. **If you want to find matches regardless of whether they match the case in your replace string, check** Case insensitive **.**
5. **By default, the search runs forward to the end of the file. Check** Search backwards **if you want the reverse.**

6. **Click the button that matches the operation you want to perform—**Find, Replace, or Replace all.
   Use Replace all judiciously. If you choose this button and then discover you've made a mistake, you can't undo the operation. Instead, you must use the Undo since save option. You can work around this by saving just before doing a Replace operation, thereby ensuring that if you do need to undo the operation immediately, you won't be losing any other work.
7. **Click** Close **when you're done searching for text.**

Undoing Edits

To undo any edits you've made since you last saved:

- **Click** Undo since save**.**
  The text window changes to reflect the last saved state.

Saving and Compiling

1. **Click** Save & Compile**.**
   Your code is compiled and checked, and either a success or a failure dialog opens to give you status.
2. **Click** OK**.**

Saving without Compiling

- **Click** Save**.**
  Your file is saved to disk, but the code is not compiled.

Running the Script

- **Click** Run script...
  The Run Test dialog opens. For instructions on how to use this dialog, see See Playing Back a Test Using Run Test in the chapter "Running Tests."

Closing the Script Editor

To close the window:

- **Click** Close**.**

Running Tests 9

You play back scripts and tests either by running JavaStar from the command line and supplying arguments that define playback, or by navigating to the Record / Playback window from the JavaStar main menu. To get to this window from the main menu, you select either Run Test or Create Script.

The Create Script window is for developing new scripts. Sometimes this involves playing back the script under test, or loading and playing a previously recorded script to bring the application or applet to a necessary state. When you're writing and debugging a test, it makes sense to do this in one place. But, in general, it's not a good place to do formal test runs. The log file contains *everything* that has happened since the Record / Playback window was launched (recordings, multiple playbacks, declaration generations) and so it's not suitable for analyzing formal test results.

Run Test is a clean, start-to-finish test run that logs results separate from any other process. This feature assumes that you are running a JavaStar test (a `.jst` file) or a script that can run, from beginning to end, without any advance setup or manual interaction. You can choose whether you want to see the JavaStar windows and your application while you run, or selectively hide them.

When you run JavaStar from the command line, you are, in essence, doing a Run Test operation, so all of the same requirements and options apply. For details on how to do this, see the chapter "Using Command Line Options."

> ***Note –*** You may need to include `JS.delay()` in your first script, to compensate for the start up time of the application or applet under test. To read more about this method, see wrap(Component) in the "Component and Control Classes" chapter of the *JavaStar API Reference*.

This section covers:

- Playing Back a Test Using Run Test
- Playing Back a Script from the Record / Playback Window
- Playback Tasks Available in the Record / Playback Window

9

Playing Back a Test Using Run Test

1. **From the JavaStar main menu, click** Run test**.**
   This brings up the Run Test dialog window (see Figure 9-1 on page 86). This window has three tabbed sections:

   - **General**—where you enter the name of the script or JST you want to run, along with any arguments your test requires. By default the General tab appears in the forefront.
   - **View**—where you determine which windows are displayed during the test run.
   - **Advanced**—where you supply settings for your working, results, failure directories, define additional CLASSPATH settings, etc.

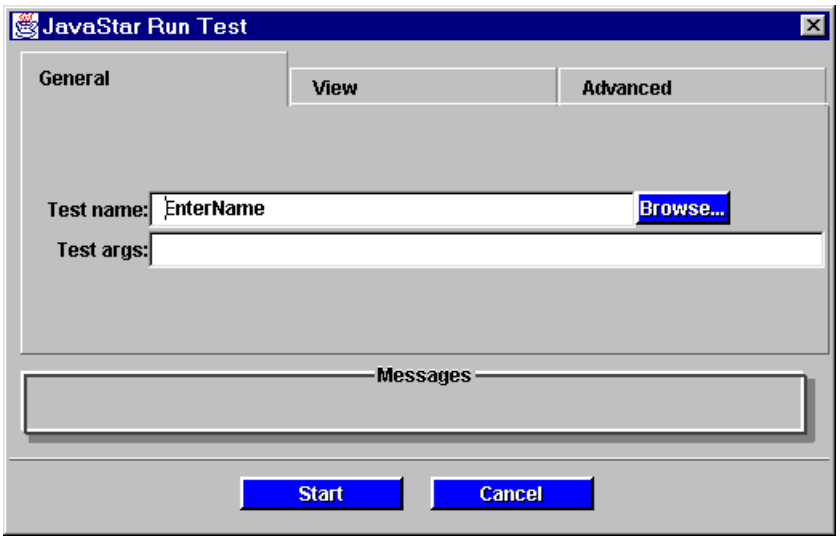

Figure 9-1 Run Test dialog

2. **Supply a test name and, if required, test arguments.**
   This is the minimum of information you must supply. When typing test arguments, be sure to put double-quotation marks around any single argument that contains whitespace or a special character. For example:

   ```
   “Alix North” 32 writer “San Francisco”
   ```

   Note – Using the Browse button, you can navigate through the directory structure and select a script (`.class` file, not `.java`) or a JST (`.jst`) to run. JavaStar inserts the name of the file you select into the Test Name field.

3. **To change the view option, click the View tab and make a selection.** Figure 9-2 on page 87 shows the Run Test window with the contents of the View tab to the forefont. If you want to change the current setting, select another options (usually Show Application and playback window by default).

   If you choose Don't show playback window, none of the JavaStar windows are displayed while playback is in process—you only see your application or applet under test. You do not have any JavaStar Record / Playback controls available to interrupt the test run. As soon as the test finishes, JavaStar closes the program under test. You need to click View Test Results in the main menu to see what happened. If, for some reason, your test throws an exception that isn't handled by your test, your program may seem to hang. You can terminate the playback process using the Status Monitor, then view results to see what problem the log file reports.

   If you choose Don't show application or playback window, neither the JavaStar windows nor your program under test is displayed during the test run. This is perhaps the trickiest selection, because it's harder to find out when your test is done. You can always use the Monitor Status option of the main menu to track progress. When the test is complete, you can view results using the results viewer.

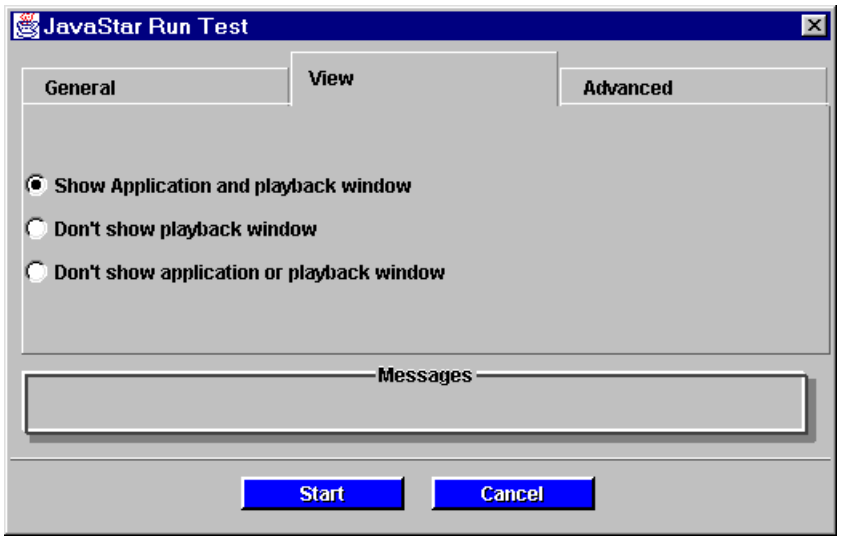

Figure 9-2 View options of the Run Test window

4. **To check your CLASSPATH setting, define a log filename, and verify that your directory settings are correct, click the** Advanced **tab to bring its contents forward.**
   Figure 9-3 on page 89 shows the Advanced tab options. The first option—Additional classpath—is perhaps the most important. This is where you specify the path to the application or applet you want to run. If this is not set correctly, JavaStar cannot start the application.

   Note – If you are using Windows and launched JavaStar by double-clicking on the icon, the Work directory, Results directory, and Jst Path are set to `..\javastar\work` (where `...` represents the directory where you installed JavaStar). This is true even if you already set options to use different directories. You can change these fields to reflect the directories you want to use, and the next time you create a script, you'll see your updated settings.

Table 9-1 Run Test Advanced options

Option	Description
Additional classpath	Any additional directories you want to add onto the existing CLASSPATH variable definition, including the path to the application or applet you are testing.
Java args	Any Java flags required by the test program. See JavaStar Command Reference for a list of valid flags.
Log filename	The log containing the results of this test run. If you don't provide a log name, JavaStar will name the file `<testname>.log` by default.
Work directory	The directory where JavaStar reads and writes scripts by default.
Results directory	The directory JavaStar uses to store the fail directory, gold directories, and log file.
JST Path	The paths JavaStar searchs when looking for JSTs and the scripts they reference.

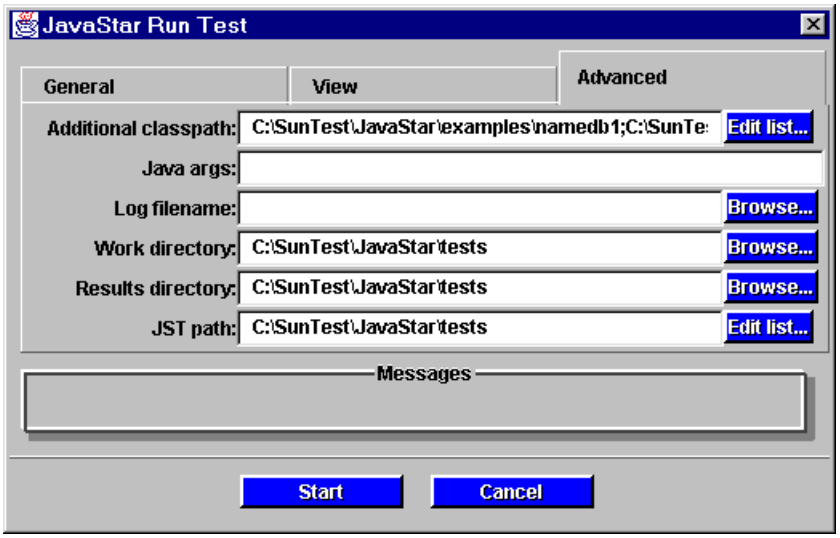

Figure 9-3 Advanced options of the Run Test window

5. **Click** Start**.**
   JavaStar opens the application or applet under test, along with the JavaStar Record / Playback window. **Th**e test begins executing immediately. If you are running a JST, Javastar also opens the JST Runner (see Figure 9-4 on page 90). This window shows the JST in graphical form. While the test executes, JavaStar flashes each node as it executes so you can follow the progress.

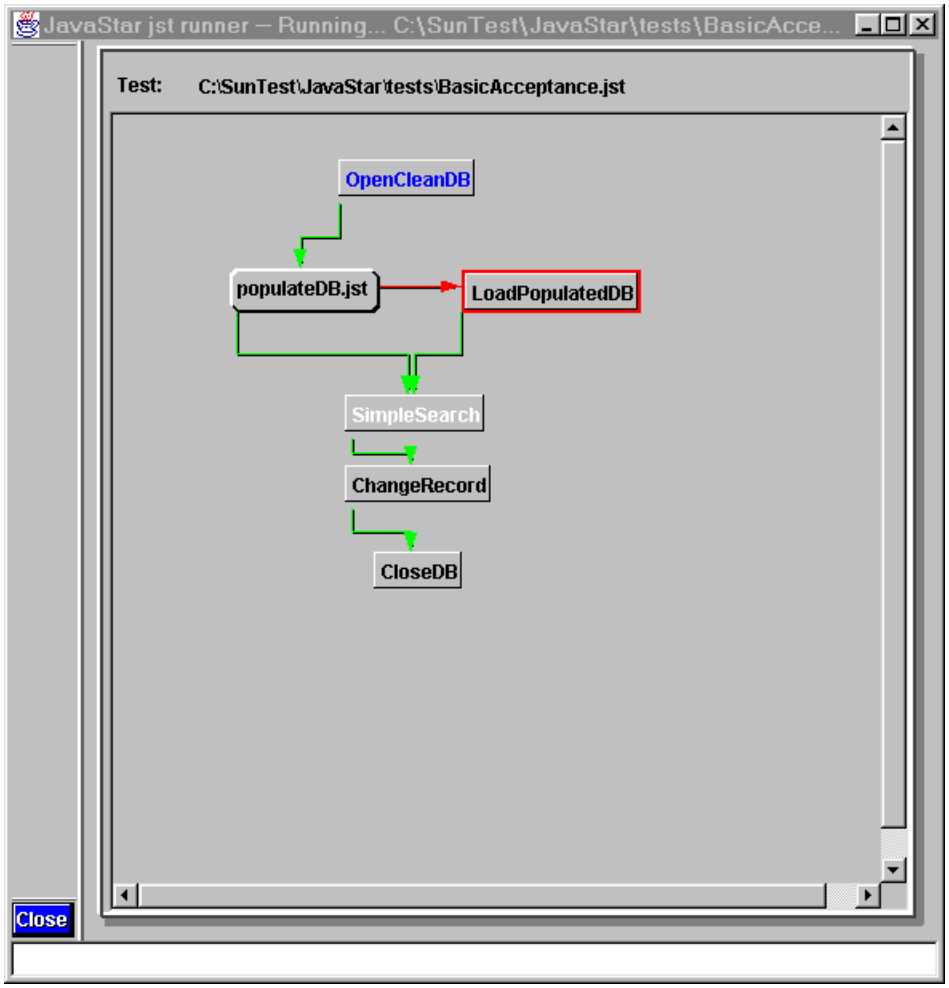

Figure 9-4 JST Runner

Note – If the JST runner encounters a node that references a script not in the CLASSPATH, testing stops and the log indicates that the class for the node could not be found.

6. **When test execution ends, click** Quit **to close all playback windows.**
If you've chosen an option that displays the JavaStar GUIs, you can review the log in the Record / Playback window before quitting. You can also quit and choose View Test Results from the JavaStar main menu, then view your results through a reporting interface.

Playing Back a Script from the Record/Playback Window

1. **From the Record/Playback window, click** Playback**.**
   This brings up the Playback Test dialog window.

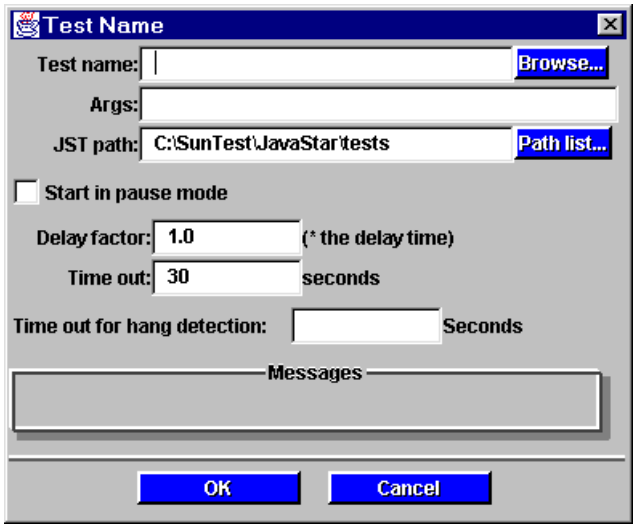

Figure 9-5 Playback Test dialog

2. **Type the name of the test to run, any arguments your test requires, and the path to use for finding JST components.**
   The test name can be the name of a script or a test (JST) file.

3. **Set the run test controls.**

Table 9-2 Controls for playing back tests

Option	Description
Start in pause mode	Starts playback but suspends it before the first event. You can then click Continue to resume playback, or click Single Step to move through the test event-by-event.
Delay factor	This is the value that JavaStar multiplies by any internal delays recorded into your test. See Setting Test Options in the chapter "Creating Project Files" for more explanation.
Time out	Maximum time allowed to continue to attempt comparison verifications. When this time value is reached, a comparison fails.
Time out for hang detection	Maximum time JavaStar will wait without a response from the application under test. When this value is exceeded, JavaStar assumes the program hanged and aborts the test.

4. **Click** OK**.**
   JavaStar loads the test. If you did not select Start in pause mode, the test begins to execute immediately.

Playback Tasks Available in the Record/Playback Window

When you playback tests with the Record / Playback window open, you have several control options available. Run Test launches the Record / Playback window, so you don't need to playback your tests through the Create Script option to get these features.

Tasks available:

- Single-stepping through a Script or Test
- Setting Options During Playback
- Inspecting Components During Playback
- Pausing, Stopping, and Quitting Playback

Single-stepping through a Script or Test

The Single Step feature allows you to step through your test script line by line. Each time you click Single Step, playback advances to the start of the next call to a static method of the class JS. Usually, this means that each click of the Single Step button executes another event (although the next call could be of another static method type).

Single stepping can be helpful when you want to debug a script or see exactly what is happening with the test program just before a failure.

> ***Note –*** If you single step to a point in your test where you invoke a modal dialog window, you will not be able to single step past this point. Use CTRL-ALT-F8, Ctrl-Shift-F8, or Ctrl-Meta-F8 to force the application to continue past the modeal window. To complete script playback, use CTRL-ALT-F9, Ctrl-Shift-F9, or Ctrl-Meta-F9. Some Sun platforms do not provide an ALT key, so if that doesn't work, try one of the other combinations. Note that the modal dialog doesn't present a problem when you playback a script without single-stepping.

1. **Click the** Single Step **button in the Record/Playback window.**
   This pauses the playback operation.
2. **Click** Single Step **for each step of the script you want to perform.**
   The script advances one event each time you click the button.
3. **To resume regular playback, click** Continue**.**
   The script continues executing without requiring user intervention.

Setting Options During Playback

The Record / Playback window provides a shortcut to changing the playback options during a test execution. This is useful if you've started playback and realize that the script is using the wrong delay factor or time-out value.

1. **In the left panel of the Record/Playback window, click the** Options **button.** The playback options are displayed in the right portion of the window. See Figure 9-6 on page 93.

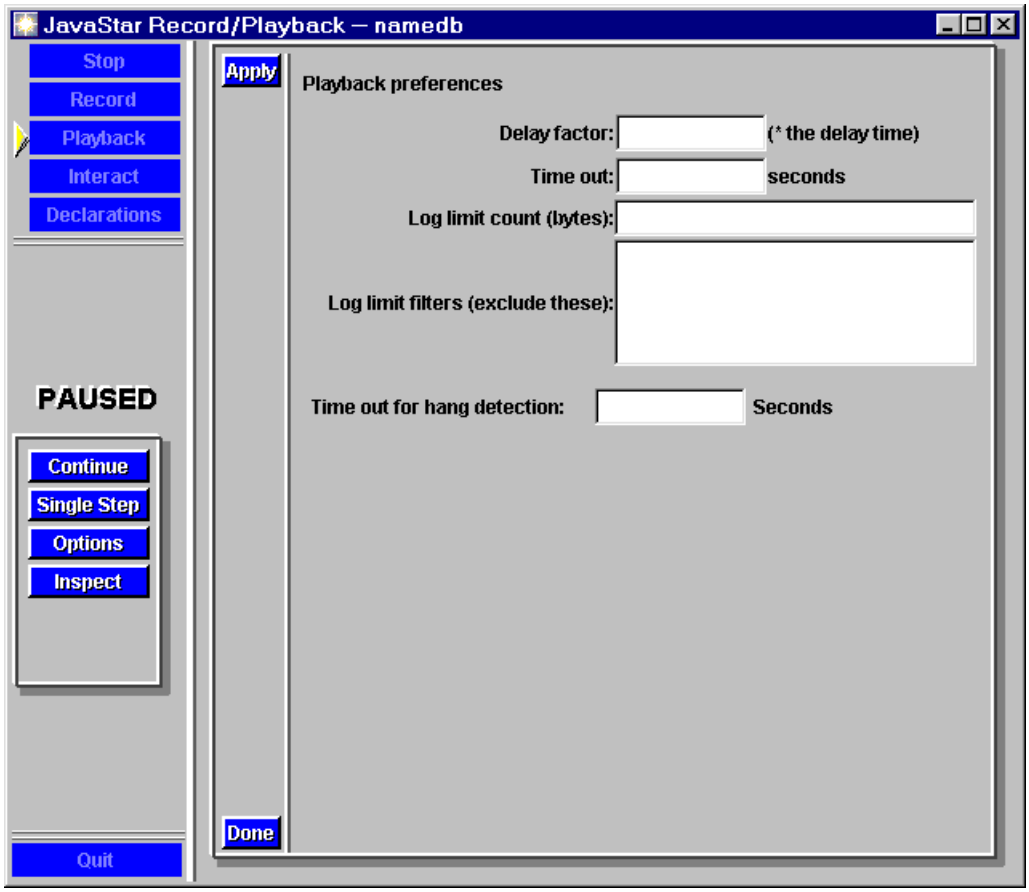

Figure 9-6 Playback options in the Record / Playback window

2. **Change the options according to your preferences.** For a detailed explanation of each playback option, see the Setting Test Options section of the chapter "Creating Project Files."
3. **Click the** Apply **button in the options panel.**
4. **Click the** Continue **button to resume recording.**

Inspecting Components During Playback

While playback is paused, you can inspect the return values for the methods and data members of any component of your test program. To do this, click the Inspect button to open the inspection panel to the right. This is the same inspection panel you can access while interacting with your application from Record / Playback. For a detailed explanation of how to use this dialog, refer to the chapter "Interacting and Inspecting."

Pausing, Stopping, and Quitting Playback

The Record / Playback window provides options for pausing, stopping, and quitting playback.

Pause suspends playback.

Stop ends test playback and requires you to reload the test script to run it again.

Quit terminates any script playback or record, closes the Record / Playback window, and exits the test program.

Monitoring Test Status 10

With JavaStar, you can have multiple tasks running at any one time—for example, you can be recording a script, running a test, and displaying the results of another test all at once. The Monitor Test Status feature of the main menu provides you with a way to monitor all currently running jobs and, in a case where your test program hangs, terminate a process.

The Status Monitor:

- Shows each activity (or job) currently running in JavaStar
- Gives the path and name of the associated log file
- Shows the time and date the activity started
- Provides a kill button to use to terminate the task

This section describes:

Viewing Details on a Process

1. **In the main menu, click** Monitor status**.**
   The Status Viewer window opens.

2. **Find the activity you want to view detail on, and click this line.**

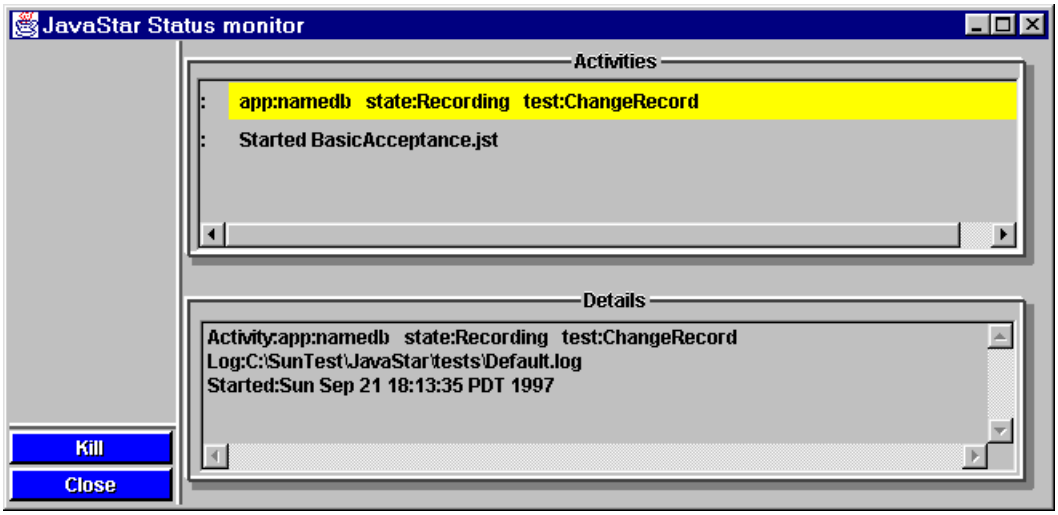

Figure 10-1 Status Monitor

The details panel displays information on the state of the activity, test name, log file name, and the start date and time.

3. **Click** Close **to dismiss the monitor.**

Killing a Job in the Status Monitor

1. **In the main menu, click** Monitor status**.**
   The Status Viewer window opens.
2. **To select the job you want to terminate, click on the job name in the Activities panel.**
3. **Click the** Kill **button to the left of the Details panel.**
   The process terminates and the item disappears from the Status Monitor.

   Note – If you kill a recording process, JavaStar saves and compiles the script before terminating the process.
4. **Click** Close **to dismiss the Status Monitor.**

Viewing and Analyzing Results 11

The JavaStar Results Viewer gives you a quick way to view the summary of a test or script run or to examine the results in more depth, filtering the files to show only the information you want to see. You can iterate through the results of a JST, examining the results node-by-node. You can jump immediately to any gold files to compare failures, view the differences, and even update the gold file as necessary.

You get to the Results Viewer by choosing View Test Results from the JavaStar main menu.

This section describes:

Anatomy of the Results Viewer

The Results Viewer is a large window with several panels (see Figure 11-1).

Before you use the Viewer for the first time, take some time to familiarize yourself with each of the buttons and panels. This section describes each of the components.

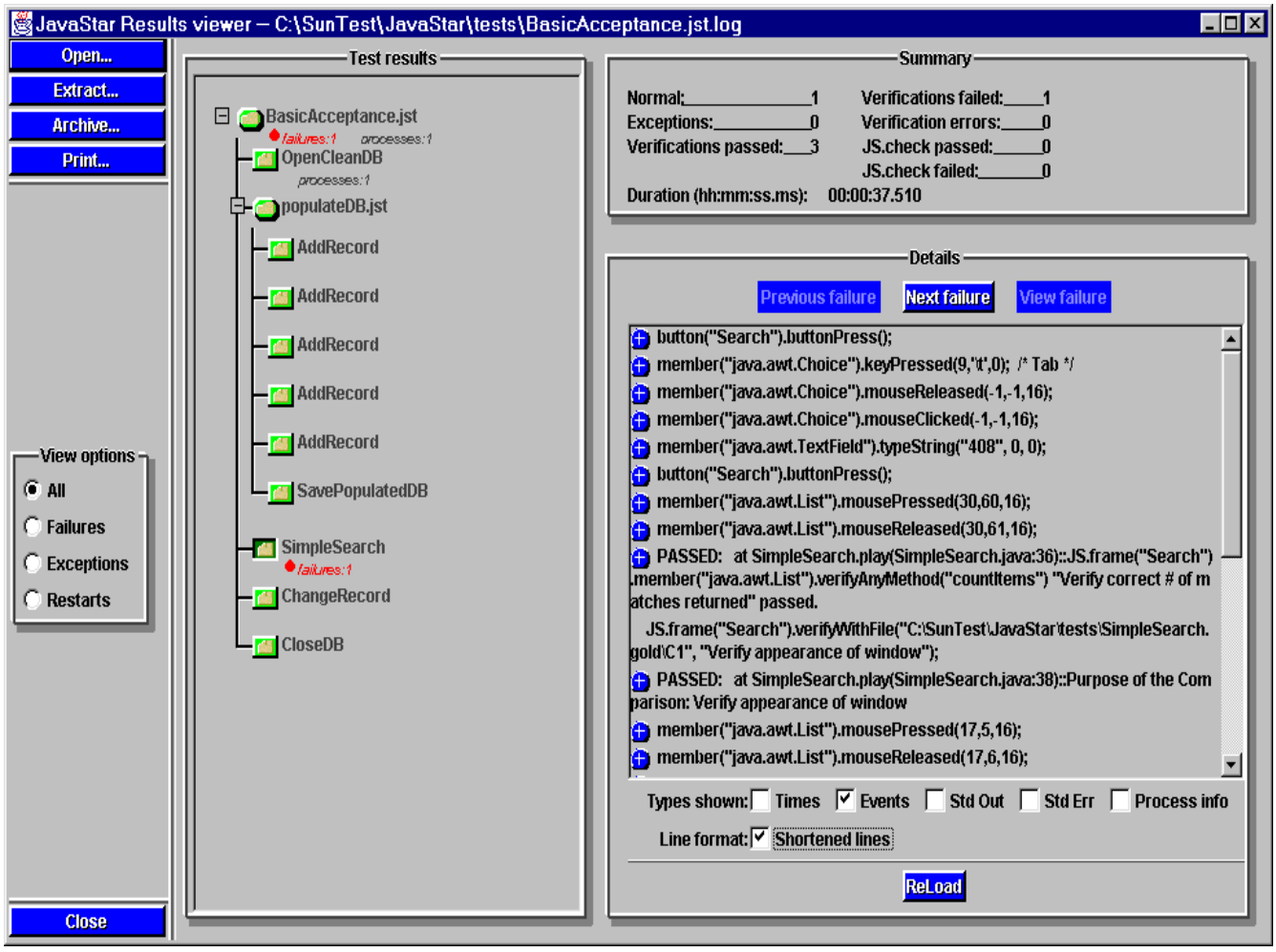

Figure 11-1 Results Viewer

Task Buttons

The Viewer task buttons are the five buttons in the far left panel, excluding the View Options selections. These represent the main functions provided by the Results Viewer.

Table 11-1 Task Buttons

Button	Use this to...
Open...	Open a `.log` file or the log from archived results, for viewing.
Extract Results...	Extract the results you want from the log and write them to an HTML or text file.
Archive Results...	Archive all results from a test run, including gold files and failure files, into a `.zip` file.
Print...	Print the current results screen.
Close...	Close the Results Viewer

View Options

These options, located in the middle of the far left panel, control the nodes you see in the Test Results panel. Only one option can be set at a time.

> ***Note –*** If you are viewing results from a single-script test run, you probably don't need to use View Options, as your test consists of a single node.

Table 11-2 View Options

Option	Sets test results to display...
All	All of the nodes in the JST. This setting is the default.
Failures	Only the nodes that contain failures.
Exceptions	Only the nodes that ended by throwing exceptions.
Restarts	Only the nodes set to restart the application or applet. By its nature, the root node is always considered a restart node.

Test Results

This is the panel showing each node of your JST in tree format. If you ran a single script, you'll see only one node.

Any node with a plus sign (+) proceeding it can be expanded to reveal child nodes. You can expand a node just by clicking on the plus sign—the plus converts to a minus (-) and all the child nodes appear below the JST. You can also collapse an expanded node by clicking on the minus sign.

If a node contained a failure, the text `failures:` followed by the number of failures appears below the node. This gives you an idea of which results you might want to explore in more detail.

Summary

This panel offers a summary of results for the node you have currently selected. If you select the a JST node (such as the first node), the Summary panel includes data from the child nodes.

The Summary panel reports on:

- The number of JST nodes that ended normally. A node that contains verification or `JS.check` failures, but which does not throw an exception, is considered normal.
- The number of JST nodes that ended by throwing an exception.
- A count of the verifications that passed.
- A count of all verification failures.
- A count of any verification errors that occurred. Verification errors are not failures, because they indicate that the verification couldn't be performed—such as: can't find gold file for comparison.
- The number of `JS.check` method calls that passed. (`JS.check` is a method in the JavaStar API that you can use to manually insert your own comparison code into a script.)
- The number of `JS.check` method calls that failed.
- The duration of the test run.

Details

The Details panel, located to the middle and lower right of the window, provides:

- A panel showing the log results pertaining to the currently selected node. This display reflects the options you set in View Options and with the buttons below this panel.
- Three buttons for navigating to failures—these are only enabled when the node you selected contains a failure. (See Table 11-3).

Table 11-3 Failure navigation buttons

Button	Action...
Previous Failure	Moves back to the previous failure.
Next Failure	Jumps to the first failure or to a failure subsequent to the current one.
View Failure	Opens up the Gold File Manager, as well as the gold file and the comparing image that failured. The Gold File manager is covered in detail in Viewing Comparison Failures and Updating Gold Files.

- A series of check boxes that you can toggle on and off, affecting the types of information displayed and the format (See Table 11-4).

Table 11-4 Type and format check boxes

Check box	When toggled on...
Times	Includes timestamps.
Events	Includes each event call made within the script
StdOut	Includes any messages sent to StdOut.
StdErr	Includes any messages sent to StdErr.
Process Info	Includes the system information from the test machine, playback options in effect, the start and end time of the test, and any
Shortened lines	Sets each line to use a terse format. Each line that has been shortened is prefaced by a plus sign in a circle. You can click on this to expand the line for more details about the event.

- ReLoad button—this is the button you need to click each time you change the settings for type or format. Reload opens the log file again, filtering for the information you want.

11

Viewing Results

1. **From the main menu, click** View Test Results**.**
   The JavaStar Results Viewer opens. By default, the name of the log file most recently generated appears in the Log file field.
2. **Click** Open...**.**
   A file navigation window opens up.
3. **Navigate to the log file or archive file you want to use and select it.**
   The filename you choose must end with a `.log` or `.zip` extension.
4. **Click Open.**
   JavaStar loads the log file and selects the first JST node in the tree. Because the first node is selected, the Summary panel shows totals for the entire test run, not just a single node.
5. **Expand nodes and log information to get the information you want.**
   JavaStar displays your JST file in tree format, with all nodes collapsed. Plus signs next to the node name indicates that the node is a JST with child nodes. To expand the node, click the plus.

   When you click on a child node (one that represents a single script), both the Summary and Detail panels update to reflect totals and log file details specific only to this node.
6. **Adjust the View Options and Detail panel check boxes to reflect the information you want to see.**
   See Table 11-2 for an explanation of each view option, and Table 11-4 for descriptions of Detail panel check boxes.
7. **If you made any adjustments to the Detail panel check boxes, click** ReLoad**.**

Viewing Comparison Failures and Updating Gold Files

Within the Results Viewer, you can view gold file comparison failures. You can examine the results to see if a failure is appropriate or if the new gold file should replace the comparison file. This procedure takes you though the process of locating a failure, viewing it, and updating the gold file with the new information.

1. **Within the Results Viewer, load the log file that contains the failure.**
   As an example, Figure 11-2 shows a portion of the Results Viewer for a simple test containing a failure. Note the `Fail:1` remark below the node name, and the failure information in the log file view.
2. **Select the node containing the failure or failures.**

3. **In the Detail panel, click the** Next Failure **button to jump to the first failure.**
   JavaStar scrolls to the failure and displays the text in red. If shortened lines is turned on, you can expand the failure for more detail by clicking on the line.

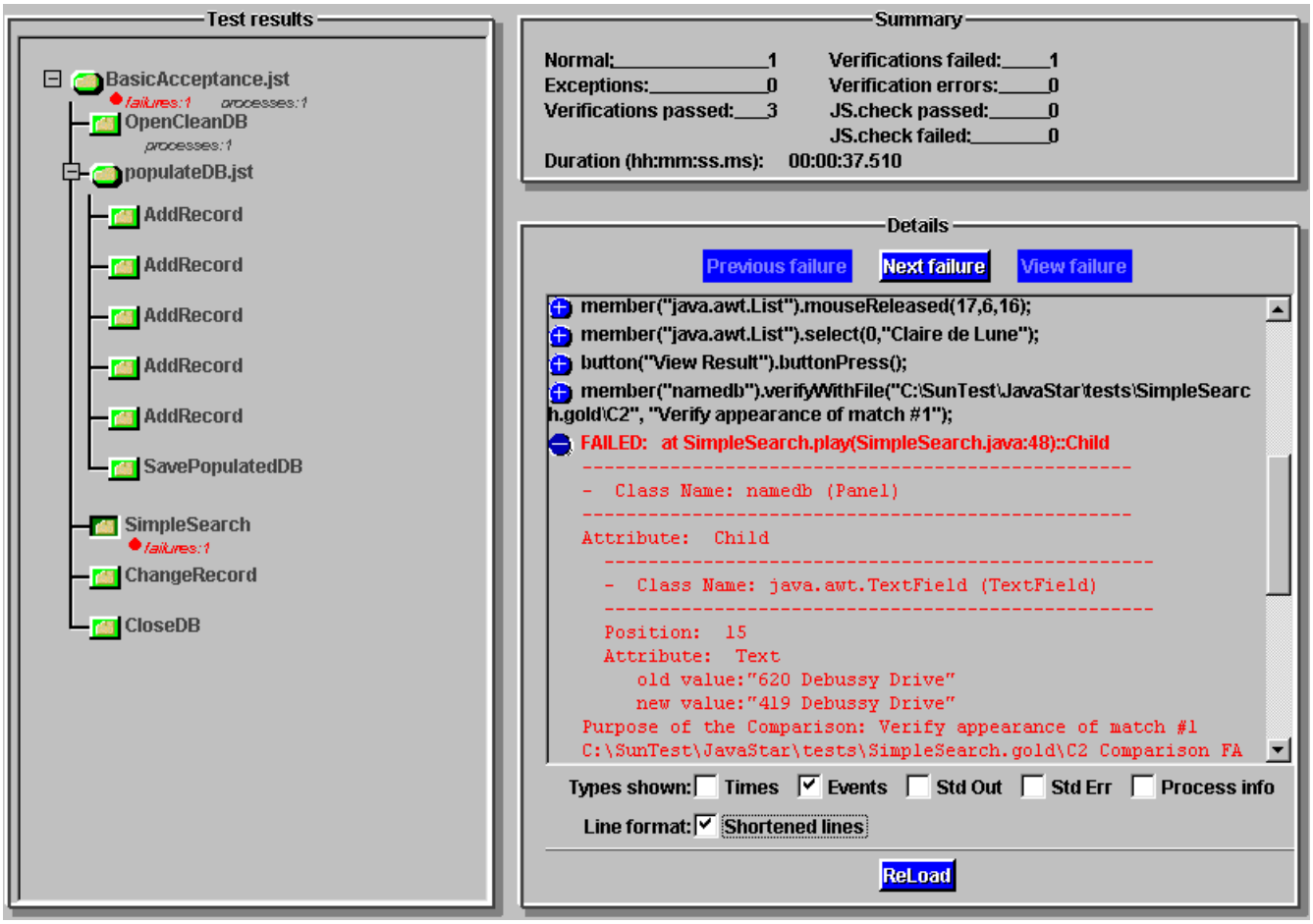

Figure 11-2 Results Viewer showing a tests that contains a failure

4. **Click** View Failure **to bring up the Gold File Manager.**
   The Gold File Manager window opens, along with a window showing the gold file (labeled Gold) and one showing the actual test results (labeled New).

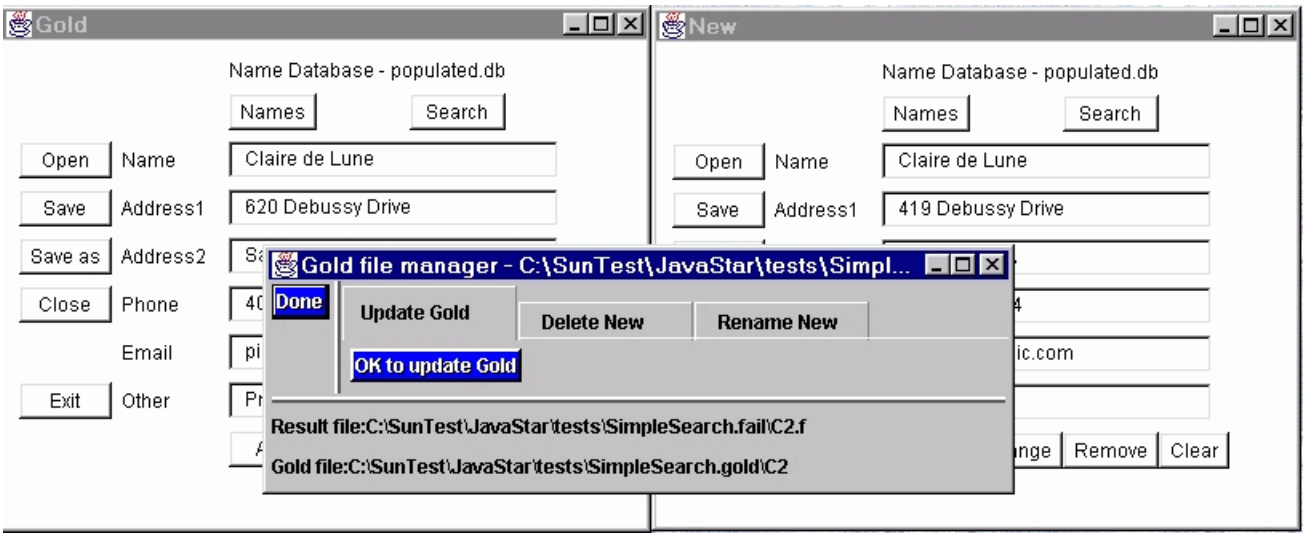

Figure 11-3 Gold File Manager with Gold and New windows

The Gold File Manager gives you three options:

Table 11-5 Options for the Gold File Manager

Option	Description
Update Gold	Replaces the gold file with the new attributes. This assumes that the new result is the correct one.
Delete New	Deletes the new results. Once you've noted the failure and reported on the failure, you probably don't need to keep it.
Rename New	Renames the new attribute file (as you specify) and stores the file in the gold directory for this script.

Note – Including a purpose in verification and synchronization checks makes it easier to evaluate failures when you later examine the log file.

5. **Compare the gold file to the new file to determine how to handle the failure.**
   Select the option that makes the most sense. For example, if the gold file is out of date and the new file contains the correct information, click Update Gold to replace the gold file.

6. **Click** Done **to close the Gold File Manager.**

Extracting Results

Using the Extract Results feature, you can save results to a text file or an HTML file. Because log files contain a great deal of information, this feature allows you to choose which information you want to include in your extract. Once you've extracted the information, you can use it to prepare reports or incorporate into with other extracted data.

To extract results for a log file:

1. **In the Results Viewer, open the log file you want to work with.**
   Don't worry about which options you have selected for log file viewing; these won't affect the extraction.

2. **Click the** Extract **button.**
   You can find this button in the left button panel. Clicking this button opens the Extract Results dialog.

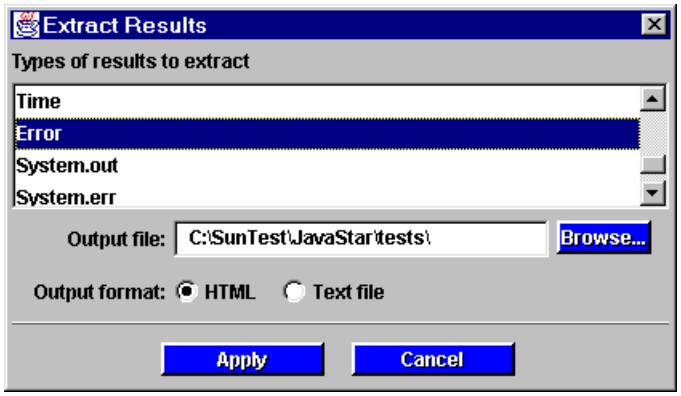

Figure 11-4 Extract Results dialog

3. **Scroll through the list and select the log features you want to extract.**
   You select a feature by clicking on the name. Note that certain features are selected by default—these are highlighted already—and you need to click on them only if you want to de-select them.

The log file options are:

Table 11-6 Log file extract options

Option	Description
Arguments	Any arguments passed to the test. These are included for each node that accepts arguments.
Control	Change of control lines—in a JST, these are lines noting that a node was entered or exited. When JavaStar leaves a node, it notes whether the condition was normal or an exception.
Other	General information.
Event	Each event the test executed.
Header	Start time, test name, playback settings, system information (details of the system environment where the test was run) and start arguments.
Result	Detailed results of verifications, synchronizations, checks, `JS.note` comments, and `JS.log` print lines.
Summary	Comparison and checking results for each node, including exit conditions.
Time	Time stamp information for each test.
Error	JavaStar internal errors and problems, such as missing scripts.
System.out	Whatever the test program outputs to `System.out`.
System.err	Whatever the test program outputs to `System.err`.

4. **Specify the path and filename you want to use.**
   By default, the path is already filled in with the work directory, but you can change this.

5. **Select** HTML **or** Text file**.**
   HTML file outputs the same information in HTML format.

6. **Click** OK**.**
   JavaStar confirms that it extracted the file information.

Archiving Results

You can archive your results—log files, gold files, and failure files—to a `.zip` file. This is useful when you want to save the complete result environment for later comparison. If you later want to examine the gold file and failure files, you can restore them. If you did not archive the files and you continued to run this test, you would lose the original information as they would be overwritten by new results.

Archive works similarly to Extract. To archive results:

1. **Open the Results Viewer.**
2. **Click the** Archive **button.**
   You can find this button in the left button panel. Click this button opens the Test Archive dialog.

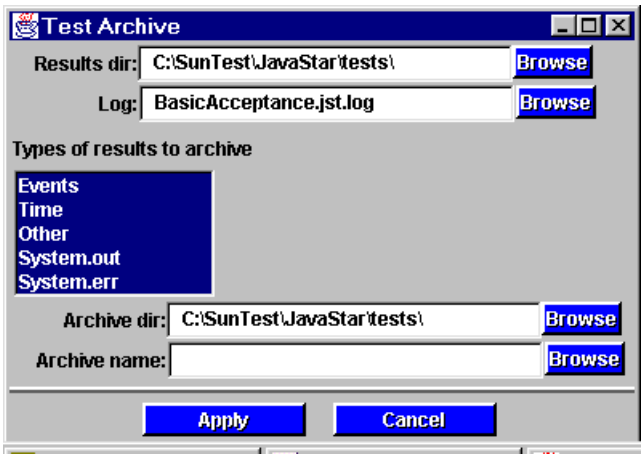

Figure 11-5 Test Archive dialog

3. **Enter the path for the directory containing the results you want to archive.**
   This is the directory you specified as the results directory when you originally ran the test.
4. **Type in the log file name or browse to locate the file.**
5. **Select which reporting options you want to include in the archive.**
   See the descriptions of these options in Table 11-6. By default, all options are selected. Click on an option to de-select it.
6. **Enter the directory where you want to store the archive.**
   If the directory does not already exist, JavaStar creates it when it generates the archive.

7. **In the** Archive name **field, type in the filename.**
   Don't provide an extension—JavaStar adds the `.zip` extension for you.

8. **Click the** Archive **button.**
   JavaStar confirms successful completion.

   Note – To later restore an archive, use a tool that restores a file compressed using the `.zip` format. You can then use the Results Viewer to view the log files and examine the restored gold files.

Printing Results

The Print feature sends an image of the Results Viewer window, as currently displayed, to the printer. Clicking Print brings up the file dialog for your printer, as you have defined it on your system.

If you want to prepare a report for printing, use the Extract option to write the information you want to a text file. You can then format that however you want and print it as a report.

Quitting the Show Results Window

- **Click** Close **to exit the Show Results window.**

Customizing Options 12

The Set Options feature of the main menu opens a multi-tabbed window where you can set GUI Options for JavaStar (options that affect how the application is displayed) and view System Info.

JavaStar writes option settings out to the `user.home` directory, to a file named `.javastar.prop` . In UNIX environments, `user.home` defaults to your home directory. In Windows 95 and Windows NT, `user.home` defaults to the directory where you have Java installed. You can check your `user.home` setting in the System Info panel.

GUI Options

This tab is where you set the options that control the look of the JavaStar application, such as font size and foreground / background colors. To set GUI options:

1. **From the main menu, click** Set Options**.**
   The Options window opens.
2. **Click on the** GUI **tab.**
   The GUI tabbed panel moves to the forefront.
3. **Adjust the settings as you want.**
4. **Click** OK.
   Your new settings are saved and the Options window closes.

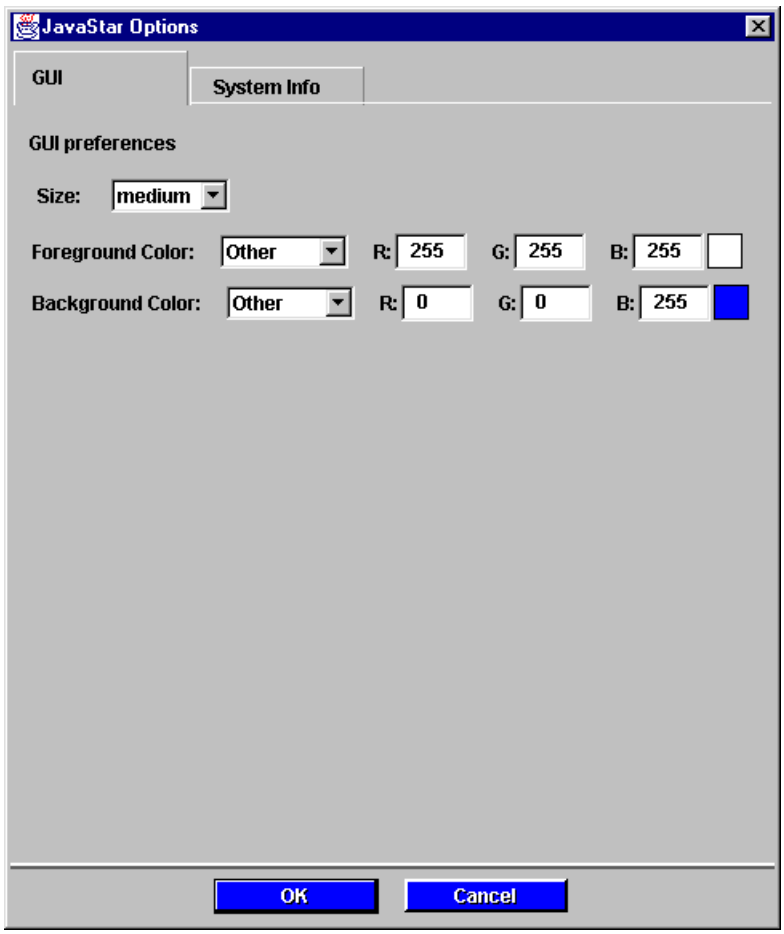

Figure 12-1 Set Options: GUI Tab

Table 12-1 GUI Options

Option	Description
Size	Sets the size of text in JavaStar windows. Choose small, medium, or large from the pop-up menu.
Foreground color	Changes the color of foreground items in the GUI, using either colors selected from the pop-up menu, or (Red, Green, Blue) values for a custom color. Choosing default from the menu resets this to the default values.
Background color	Changes the color of foreground items in the GUI. This works the same as Foreground color.
Prefixes (package/class) to ignore for source highlighter.	Specifies packages or classes as a library, so they won't be shown during playback—instead the call to the library will be highlighted.

System Info

The System Info tab is where you can view information about your Java and JavaStar setup. This panel displays the:

- JavaStar version
- Java home setting (`user.home`)
- User home directory
- Java version
- Java vendor
- OS name
- OS architecture
- CLASSPATH setting

None of these values are editable in this window—they all reflect system settings that are read as you launch JavaStar.

> ***Note –*** The System Info panel shows the CLASSPATH as it was set when JavaStar was launched. The project file extends this CLASSPATH setting with the path to the application or applet, test directories, and other extensions. The extensions do not display in this window; refer to Project Settings for more CLASSPATH information.

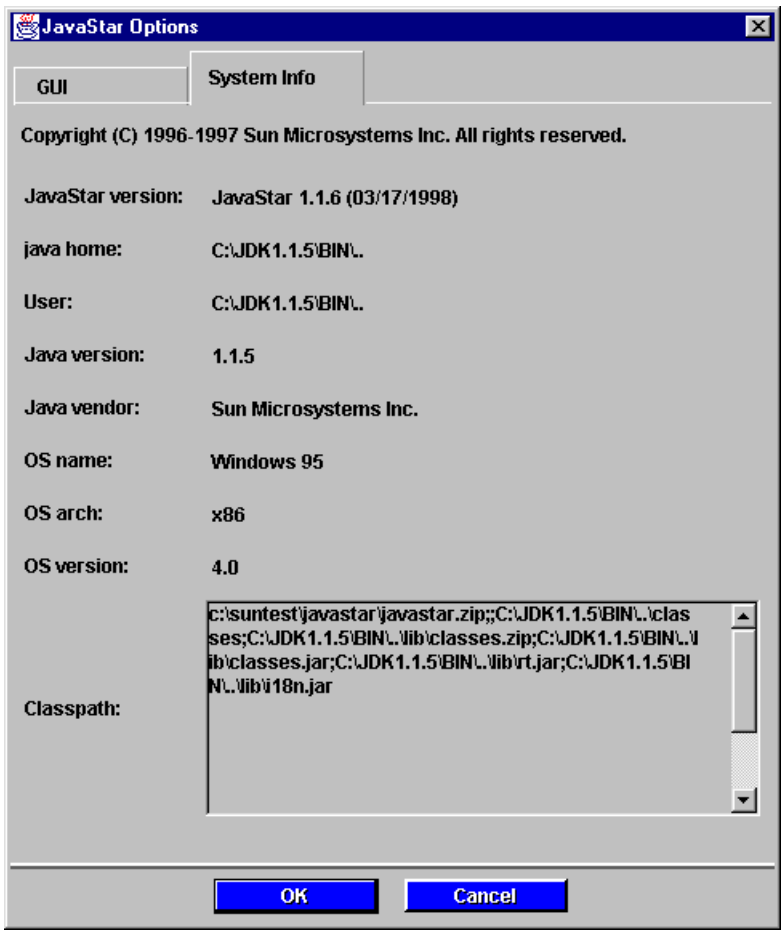

Figure 12-2 Set Options: System Info tab

Using Command Line Options 13

Each of the playback features provided by the JavaStar GUI can be set through command line flags, as well. This means that you can run tests, start recording, and process log files all from the command line. With test runs, it also means that you can run tests automatically by stringing command line calls together in a script or batch file.

Topics:

Running Tests

To run a test from the command line, you must, at minimum, provide the test name and any arguments required by the test. Beyond that, you can specify any of the options available in the Run Test dialog. Table 13-1 shows the correspondence between the GUI and the command line flags.

The order for command line options is only loosely defined. For example, Java arguments must immediately follow the `javastar` call, and the `-testargs` flag must always fall at the end, but the other options can be defined in any order.

Table 13-1 Corresponding Run Test fields to command line flags

Run Test option	Corresponding Command Line Flag
Test name	`-jst` JstName `-script` classname
Test arguments	-testargs (args)* This must be the last flag in your command line string.

Java arguments	`-J`arguments
	Preface each argument with `-J`. Do not include a space between `-J` and the argument. For example:
	`-J-prof`
	`-J-mx64m`

Log filename	`-log` filename
Work directory	`-workdir` dir
Results directory	`-resdir` dir
Jst path	-jstpath searchpath
Show Application and playback window	`-gui` (preface with `-play`)
Don't show playback window	This is the default setting, if `-gui` or `-invisible` are not specified.
Don't show Application and playback window	`-invisible` (preface with `-play`)
(none)	`-1proc` Run with with just one process. You can only use this option on tests that do not contain restart nodes or crash recovery nodes. You also need to include the correct library in the CLASSPATH—for example, if you are using JDK 1.1.4, be sure to add `javastar/lib/114.zip` to the CLASSPATH.

Environment and Playback Controls

In addition to these flags, JavaStar provides controls that correspond to the Playback Options and Environment Options available in the GUI (under Set Options), as well as some additional controls. You can use these to further define your test environment. For example, if you're running a test on a slow system, you might need to scale delays so that the test doesn't run too fast for the architecture to keep up.

Table 13-2 Command line flags that correspond to environment options

Option Setting	**Corresponding Command Line Flag**
Work directory	`-workdir` dir
Results directory	`-resdir` dir
Jst path	-jstpath searchpath
Additional Classpath	`-jcpf` flag Flag to use for the JVM in place of '`-classpath`' `-kcpf` flag Flag to use for the compiler in place of '`-classpath`'
Java virtual machine	`-jvm filepath` Specifies the Java Virtual Machine to use for subprocesses.
Java compiler	`-jc filepath` Specifies the Java compiler to use when compiling.
Properties file	`-props propertyfile` Loads user test properties from file for the playing test.

Table 13-3 Command line flags that correspond to playback options

Option Setting	Corresponding Command Line Flag
Delay factor	`-scale` n Scales the delay by a factor of n.
Time out	`-timeout` n Sets compares to timeout after n seconds.
Log limit count (bytes)	-llcount size Limits log count—provide size in characters.
Log limit filters (exclude these)	-llfilter flags Suppresses the log file content represented by the flags. See Table 13-8 for a list of log filtering flags.
Time out for hang detection	-hangtime n Sets the interval of time (in seconds) a playback script can continue without receiving a response from the application under test. When a script reaches `hangtime`, it terminates in an `AsynchTimeoutError`.

Table 13-4 Additional controls

Option Setting	Corresponding Command Line Flag
`-J<flag>`	Flag or argument to pass to JVM subprocess (e.g. `-J-mx64m`)
`-K<flag>`	Flag or argument to pass to compiler subprocess (e.g. `-K-g`)
`-jdk` version	Tell JavaStar which JDK your JVM is equivalent to. Must be one of 1.1.1, 1.1.2, 1.1.3, or 1.1.4.

Exit Codes

JavaStar provides exit codes for playback operations, shown in Table 13-5. If the test program calls `System.exit()` before the script exists, the exit code can have any value.

Table 13-5 Exit codes for playback

Exit Code	Description
`0`	Success
`1`	Verify failed
`2`	Check failed
`3`	Verify and check failed
`4`	Exception thrown
`5`	Exception thrown, verify failed
`6`	Exception thrown, check failed
`7`	Exception thrown, verify and check failed

Managing Log Files

If you already know what log file information you want from your test runs, you can set your tests to suppress any other information, or you can add a command line call to do post-processing on a log file and extract the data you need.

For log file filtering at run test time, use the options:

Table 13-6 Log filtering options at run test time

Log filter option	Description
`-log` filename	Uses the filename you specify for the log file
`-llcount` size	Limits the log to the size you specify, in characters.
`-llfilter` flags	Suppresses the log file content represented by the flags. See Table 13-8 for a list of log file filters.

For log file filtering after the file has already been generated, use:

Table 13-7 Log filtering options for existing logs

Log filter option	Description
`-logfilter` filters `logfile` [outfile]	Applies filters to the log file and prints this out either to outfile (optional) or `System.out` (default). See Table 13-8 for a list of log file filters.
`-loghtml` filters `logfile` [outfile]	Similar to `-logfilter`, this applies filters to the logfile and writes the filtered log out in HTML format. The filter may include the characters `E`, `T`, `1`, `2`, or just `-` to indicate no extras. Additional log types are limited to Event, Time, Stdout(1) and Stderr(2).
`-logsum` (logfile)+ [`-out` outfile]	Summarizes the log file (or files) and prints it to outfile or to `System.out`. This must be the last flag in the command line.
-archive archivefile [logfile]+	Adds the log files into the archive. Log files already in the archive are overwritten. This must be the last flag in the command line.

The `-llfilter` and `-logfilter` commands use log filtering flags—these are the flags that appear as the first character in every line of the long file.

Table 13-8 Log filtering flags

Log filter flag	Description
`A`	Arguments (arguments passed to each test)
`C`	Control (flow of control within the test)
`D`	Other
`E`	Event (each event executed)
`H`	Header (system information)
`M`	Machine
`R`	Result

Table 13-8 Log filtering flags

Log filter flag	Description
S	Summary (comparison and checking results)
T	Time (timestamps)
X	Error
1	System.out
2	System.err

For a single reference on all command line flags, see JavaStar Command Reference in the *JavaStar API Reference*.

13

Using JavaStar with HotJava 14

This chapter describes how to test applets within the HotJava Browser. It covers:

Installing the HotJava Browser

You must install the HotJava Browser on your system. You can find the product at the Java web site:

```
http://java.soft.com
```

When this guide refers to <HotJava Installation>, please substitute the full path to the directory in which you installed HotJava, for example "`d:\HotJava`" or "`/usr/local/HotJava`".

Setting Up a Project for HotJava

Once your browser is installed, you can create a project that invokes the browser as an application through JavaStar. You then start your applet from the "Record / Playback" window.

Here are the settings required for HotJava:

HotJava Application

The class that runs HotJava is `sunw.hotjava.Main`. It requires access to all HotJava classes in the <HotJava Installation>`/lib`.

On the App pane:

1. **Set the application class to `sunw.hotjava.Main`**
2. **Set the application classpath to include `<HotJava Installation>/lib` and these archives found in that directory:**

- `classes.zip`
- `ssl.jar`
- `x509v1.jar`

HotJava Java Environment

Change the Java Environment of the project to include these arguments to the Java Virtual Machine:

- `-ms4m`
- `-mx40m`
- `-Dhotjava.home=<HotJava Installation>`

Recording a Test

When you create a test script, JavaStar starts the browser. It takes some time for the browser to load. As the first part of your recording, open the applet that you wish to test. Then you can access all the applet components. As HotJava is itself a Java application, you can also verify the contents of the status area and other parts of the browser.

Using JavaStar with Java Plug-in 15

This chapter describes how to use the Java Plug-in to test applets under Netscape Navigator or Microsoft Internet Explorer. It covers:

Issues to Consider When Testing with the Java Plug-in

When using JavaStar with the Java Plug-in, keep in mind:

- You must convert your HTML code to use the Java Plug-in
- To use the Results Viewer to analyze results, you need to run JavaStar separately from the browser.
- You can only capture actions controlled by the Applet, not the browser. For example, clicking the Back button might take you to the previous page when you are recording, but that action won't be recorded in the test script.

Installing Applications to Use the Java Plug-in

The order in which you install your browser, the Java Plug-in, and JavaStar are important.

1. **Install Netscape Navigator or Internet Explorer.**
2. **Install the Java Plug-in.**
   Please go to http://java.sun.com/products/index.html, download Java Plug-in, and install it on your system.
3. **Install the JavaStar for Java Plug-in.**
   To do this:
   a. **Download the file Java Plug-in file.**
      For Solaris, download `JavaStar-JPI-117.tar.z`.

      For Windows 95/NT, download `JavaStar-JPI-117.zip`.

b. **Uncompress the file in the directory where you want the Java Plug-in directory to reside.**
If you are using Solaris, uncompress the file using the commands:

```
uncompress JavaStar-JPI-117.tar.Z
tar -xpf JavaStar-JPI-117.tar
```

For Windows 95/NT, use an unzip program to uncompress `JavaStar-JPI-117.zip`.

4. **Install JavaStar.**

5. **Replace the Java Plug-in file `rt.jar` with the file of the same name provided with JavaStar.**

a. **Locate the `rt.jar` file in the Java Plug-in directory and change the name to create a backup. For example, rename `rt.jar` to `rt.jar.orig`.**
If you are running Solaris, you should find `rt.jar` in the directory:

```
$HOME/.netscape/java/lib
```

In Windows 95/NT, default location for the Java Plug-in directory is

```
c:\Program Files\Java Plug-in 1.1\lib.
```

b. **Copy the `rt.jar` file from the JavaStar directory to the Java Plug-in directory.**
Under Solaris, copy `Solaris/117/rt.jar` to `$HOME/.netscape/java/lib.`

If you are using Windows 95/NT, copy rt.jar from `Windows\117\rt.jar` to `c:\Program Files\Java Plug-in 1.1\lib`.

Converting Your HTML to Use the Java Plug-in

Before you can use the Java Plug-in, you need to modify the HTML for your applet. You can use the Java Plug-in HTML Converter to make these modifications. Refer to the Java Plug-in documentation for details.

Testing with the Java Plug-in

1. **Start the JavaStar file server.**

```
java javastar -fserve
```

Note: this should be started in the directory containing the tests, and with the classpath set to javastar/javastar.zip.

2. **Open `javastar_jpi.html` page in Netscape Navigator or Internet Explorer.**
   You can find `javastar_jpi.html` in the JavaStar directory.

   The JavaStar Record / Playback window opens. This window is identical to the Record / Playback window you see when running JavaStar outside of Navigator, except that there is no Quit button. To exit JavaStar, you need to exit Navigator.

3. **In the browser, load the applet you want to test.**

4. **Record or playback a test.**

   Note – If you are running over a network and your test requires that you open a file, this may cause a security violation.

5. **When you are done testing, quit the browser.**

15

Locators for Non-Components 16

JavaStar directly supports all Java AWT-based components. An AWT-based component is an object that extends `java.awt.Component`. Some toolkits, while written completely in Java, do not use the AWT event model and do not extend `java.awt.Component`. JavaStar considers these objects to be *non-components*.

This chapter explains the difference between components and non-components, what you need to provide to JavaStar to test non-component objects, and what tools are available to help you. Specifically, this chapter covers:

- Recording Tests with Non-Components
- Locators as Non-Component Support Modules
- Implementing a Locator for a toolkit
- Referencing Locators in JavaStar
- Using the API with Non-Components

Recording Tests with Non-Components

Without non-component locators, recordings on non-components can break relatively easily. For example, clicking on a button in the Widgets.gui example (created using the Marimba Bongo toolkit) without using any non-component support produces the following JavaStar code:

```
JS.frame("Example Widgets").member("PlayerPanel").
multiClick(71,66,16,1);
```

This statement clicks on a screen location in the Bongo frame. This works, as long as the Bongo layout manager doesn't move any objects around. If the layout does change, the button will not be at recorded screen location (71,66) and the script will end in an exception.

When you record the same action using the non-component support module, the same JavaStar recording produces code that clicks on a specific object, not a screen location:

```
JS.frame("Example Widgets").member("marimba.gui.PlayerPanel").
getNonComponent("bongo","PopupWidget.Presentation%0.
FolderWidget%0.PageWidget%1.GroupBoxWidget%3.GroupWidget%0.
CommandButtonWidget%0").multiClick(5,10,16,1);
```

This code uses the benefits of object technology, and is much more robust. Now minor screen adjustments do not break existing test suites.

Locators as Non-Component Support Modules

The non-component support module mentioned in the previous description is called a *locator*. A locator is an object that contains information that JavaStar uses when referencing a non-component object. Locators have two methods—one for recording and one for playback. When you are recording, JavaStar sends a screen location to the locator. The locator looks through all objects, locates the object in that location, then returns a string to insert into the script. During playback, JavaStar sends locator strings back to the locator, and the locator returns the screen location.

JavaStar currently provides locators for several popular non-AWT toolkits:

- Marimba™ Bongo™
- Netscape Internet Foundation Classes (IFC)
- JFC Swing
- KL Group

You can find the class files and the source code for the Bongo, IFC, and KL Group locators in the `../javastar/contrib/locators` directory. For KL Group, `newKLG` is the most recent version.

You can find the JFC Swing locator ‘ in the `../javastar/contribe/jfc` directory (`newNCL` is the latest locator).

JavaStar provides a simple and a full version of each locator. The simple version allows identification only by index. The full version allows internal names and support for object captions, as well as indexes.

If these are the only non-AWT toolkits your applications or applets use, you do not need to create a locator—you just need to know how to use the existing ones. For other non-AWT toolkits, however, you do need to implement your own locators. This requires knowledge of Java, but JavaStar provides tools and examples so that you don’t have to create this from scratch.

Implementing a Locator

Writing a locator consists of implementing the `JSNonComponentLocator` class of the JavaStar API. This class contains two methods: `findObject()` and `getNamed()`. For information on this class, see JSNonComponentLocator in the *JavaStar API Reference*. You can also use the source code for the provided locators as examples.

You can find more on implementing locators in the tutorial lesson, Writing Non-Component Locators.

> ***Note –*** To make new locators easy to find, keep them inside the existing locator package.

Referencing Locators in JavaStar

If your application or applet requires a locator, you can include the names of the locators in your project file. For instructions, see the section "Defining Locators, Declaration Classes, and Text Map Classes" in the chapter "Creating Project Files."

You can also define locators in the Record Test Script dialog that is displayed when you begin recording a script. You can type the package.class name in or choose the locator you want from a list.

Typing the Locator into the Field

1. **In the Record/Playback window, click Record.**
   The Record Test Script dialog opens. See Figure 16-1.

2. **Type the name of the locator class you want to use into the** Non component locators field**.**
   If you have more than one locator, use the Locator list. function to build a list.

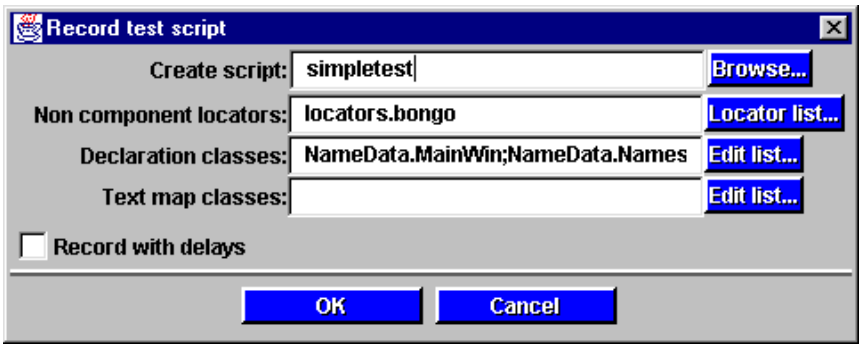

Figure 16-1 Record Test Script dialog with a locator provided

Using the Locator List

1. **Click the** Locator list **button in Record Test.**
   A Select Non-Component Locators dialog opens. See Figure 16-2.

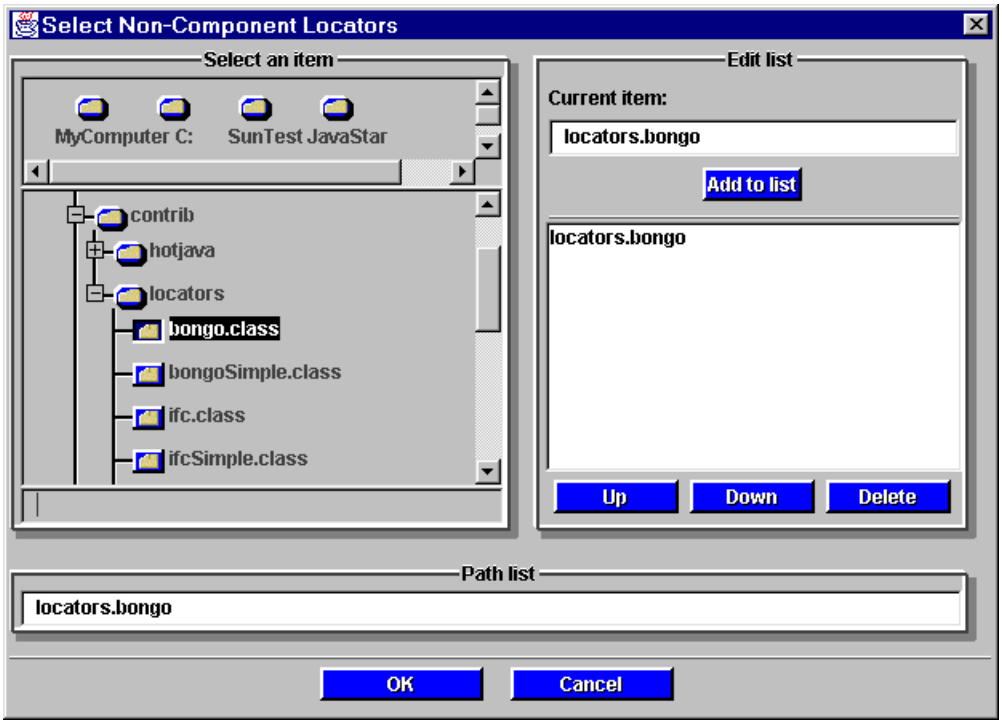

Figure 16-2 Locator List

2. **In the Select an item panel, navigate to the `\contrib\locators` directory and open the directory.**

3. **Select the name of the locator class you want to use.**
   The locator appears in the Edit list panel, under Current Item.

4. **Add the locator to the list by clicking** Add to list**.**
   The locator now appears in the Path list panel and in the list box of Edit list. If you have multiple locators listed, you can move then up or down on the list, or delete them, using the buttons at the bottom of the Edit list panel.

5. **Click** OK**.**

Using the API with Non-Components

The `JSNonComponent` class of the JavaStar API provides many of the same methods as the `JSComponent` class, but is specifically designed for use with non-components. `JSNonComponent` does not provide all of the functionality of JSComponent, but it does provide equivalents for most of the simple methods of JSComponent.

For example, using `JSNonComponent` you can perform mouse actions on a non-component (`mouseClicked()`, `mouseMoved()`, `mouseReleased()`, etc.). You can also use a simple version of `verifyAnyField()` and `verifyAnyMethod()` as well as other convenience functions. For more details, see the "Non-Component Classes" chapter in the *JavaStar API Reference*.

> ***Note –*** JavaStar supports "verify with simple method / data member" for non-component. However, "Verify with gold file" is not supported. You can only compare simple methods and data members.

16

Text Map Classes 17

Text map classes tell JavaStar how to map components to text names. In general, text maps are user-defined. You can use a text map for applications and applets where JavaStar cannot use `setName()` to extract a meaningful text name to use in the test code.

> ***Note –*** You are not required to write a text map class for components that do not have names. JavaStar will function fine without them; the test code and results might just be a little cryptic.

What Text Maps Are

In JavaStar, a text map class provides the information JavaStar needs to extract a text name for a component that otherwise does not provide one. Components that do not provide text names are:

- Bit-mapped image components (for example, components that extend Canvas or Panel)
- Lightweight components of your own design
- Any components that do not use the `setName()` method to associate a name

The code JavaStar generates for these components can appear cryptic. For example, you might see

```
member(“MyClassName”, 3)
```

where the class name refers to the component, and the number that follows indicates the position within the containing object. When you edit a test, or when you try to interpret test results, this might not be helpful.

With a text map, however, you tell JavaStar how to associate the component with a text name, so that instead you would see:

```
member(“MyClassName”, “Exit”)
```

where `Exit` is the text name you provided.

17

How to Write a Text Map Class

A text map class implements the JavaStar API `JSTextMapping` interface, providing code for the `computeText()` method. The only requirements of this method are that it accepts a Component as a parameter, and that it returns a String containing the text name (or null).

How you determine the text name depends on your program. This example shows how a text map for Swing (part of the Java Foundation Classes) looks:

```
package jfc;

import suntest.javastar.lib.JSTextMapping;
import java.awt.Component;

public class JfcTm implements JSTextMapping {

   public String computeText(Component c){
      if(c instanceof com.sun.java.swing.AbstractButton){
        return ((com.sun.java.swing.AbstractButton)c).getText();
      }
      return null;
   }
}
```

Troubleshooting 18

For up-to-date troubleshooting information, see the JavaStar Frequently-Asked Questions document on the SunTest web site -- http://www.sun.com/suntest/JavaStar/FAQ.html.

JavaStar Command Reference A

This appendix provides basic reference information on:

- JavaStar directories
- JavaStar Command Line Arguments

JavaStar directories

JavaStar references several directories while running:

Directory	Description
Work directory	This is where JavaStar stores (by default): • The scriptname`.java` and scriptname`.class` files for scripts you create • A scriptname`.gold` subdirectory, where JavaStar keeps any gold files (master comparison files) you generate The `ST` directory, a SunTest proprietary directory
Results directory	This is where JavaStar creates: • The log file for script results • A `fail` subdirectory, where JavaStar keeps the failed comparison files from the most recent playback
JST Path	This is a series of directory paths you define to tell JavaStar where to look for JST components.

The work directory and results directory both default to the directory where you launch JavaStar. The JST path is set to the work directory by default.

You can permanently override these defaults by setting your Setting Test Options. JavaStar also allows you to override these directory settings for specific processes.

Note – JavaStar requires the `ST` directory to perform its functions, but after you exit JavaStar or finish any JavaStar tasks you run from the command line, you can delete this directory.

JavaStar Command Line Arguments

Table 18-1 JavaStar command line switches

Command Line Flags	**Description**
`-usage`	Displays a list of available command line options for JavaStar
`-help`	Same as `-usage`.
`-sysinfo`	Shows system properties
`-version`	Displays the version number of JavaStar
`-logfilter` flags `logfile` [outfile]	Applies flags as filters to the log file and prints this out either to outfile (optional) or `System.out` (default). See Table 18-2 for a list of log file filters.
`-logsum` (logfile)+ [`-out` outfile]	Summarizes the log file (or files) and prints it to outfile or to `System.out`. This must be the last flag.
`-loghtml filter infile [outfile]`	Like logfilter, except extracts to HTML and filter is limited to "ET12"
`-archive` archivefile [logfile]+	Adds the log files into the archive. Log files already in the archive are overwritten. This must be the last flag.
`-jst` name.jst	The JST file name to use for editing or playback.
`-script` classname	Name of the script to use for recording or playback. This must be a fully-qualified class name.
`-props` propertyfile	Loads user test properties from propertyfile and uses these during playback.
`-llfilter` flags	Suppresses the log file content represented by the flags.
`-llcount` size	Limits log count—provide size in bytes.

Table 18-1 JavaStar command line switches

Command Line Flags	Description
`-workdir` dir	Uses dir as the output directory for recorded scripts, generated declaration classes, and as the default directory to search for scripts, declaration classes and JSTs.
`-resdir` dir	Uses dir as the root directory for all fail files.
`-jstpath` searchpath	Defines searchpath as the path JavaStar uses to locate components of JSTs. This uses the same syntax as `CLASSPATH`.
`-log` filename	Writes the log to filename.
`-scale` n	Scales the delay by a factor of n.
`-timeout` n	Sets compares to timeout after n seconds.
`-hangtime` n	Sets the interval of time (in seconds) a playback script can continue without receiving a response from the application under test. When a script reaches `hangtime`, it terminates in an `AsynchTimeoutError`.
`-jvm` filepath	Sets the Java Virtual Machine to use for subprocesses.
`-jc` filepath	Sets the Java compiler to use when compiling scripts.
`-jcpf` flag	Flag to use for the JVM in place of `-classpath`.
`-kcpf` flag	Flag to use for the compiler in place of `-classpath`.
`-J` flag	Flag or argument to pass to JVM subprocess (e.g. `-J-mx64m`)
`-K` flag	Flag or argument to pass to compiler subprocess (e.g. `-K-g`)
`-jdk` version	Tells JavaStar which JDK your JVM is equivalent to. Must be one of 1.1.1, 1.1.2, 1.1.3, or 1.1.4.

Table 18-1 JavaStar command line switches

Command Line Flags	Description
`-applet [html]+`	Record all the applets in all the HTMLs at once. If you use this option, it must be the last flag.
`-app` classname `[args]*`	A record option. Use this classname as the program under test and send these arguments to that program. If you use this option, it must be the last flag.
-fserve	Starts up JavaStar file server to work with JavaStar for Java Plug-in.
`-play`	Sets JavaStar to playback mode.
`-1proc`	Run the playback in the current process rather than a subprocess. (Requires `-play`) You can only use this option on tests that do not contain restart nodes or crash recovery nodes. You also need to include the correct library in the CLASSPATH—for example, if you are using JDK 1.1.4, be sure to add `javastar/lib/114.zip` to the CLASSPATH.
`-gui`	Shows the JavaStar Record / Playback GUI during playback.
`-invisible`	Hides the JavaStar Record / Playback GUI during playback.
`-testargs` [args]*	Sends the args as arguments to the test in playback. If you use this option, it must be the last flag.

Table 18-2 Log filtering flags

Log filter flag	Description
A	Arguments (arguments passed to each test)
C	Control (flow of control within the test)
D	Other
E	Event (each event executed)

Table 18-2 Log filtering flags

Log filter flag	Description
H	Header (system information)
M	Machine
R	Result
S	Summary (comparison and checking results)
T	Time (timestamps)
X	Error
1	System.out
2	System.err

Index

D

E

G

I

J

S

T

V

W