

EXPERIMENT-6

AIM:study of Win Runner testing tool and its implementation

a. win Runner testing process andwin runner user interface

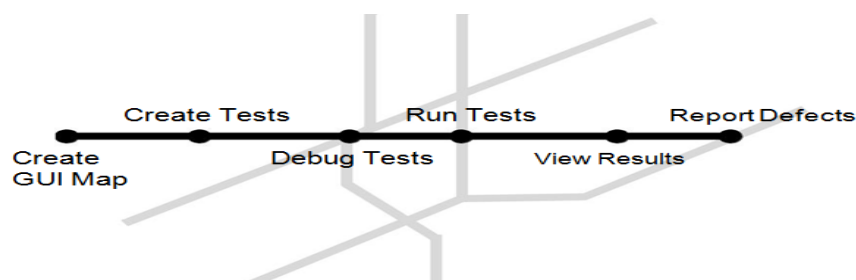
Introduction

Welcome to WinRunner, Mercury Interactive's enterprise functional testing tool for Microsoft Windows applications. This guide provides detailed descriptions of WinRunner's features and automated testing procedures. Recent advances in client/server software tools enable developers to build applications quickly and with increased functionality. Quality Assurance departments must cope with software that has dramatically improved, but is increasingly complex to test. Each code change, enhancement, defect fix, or platform port necessitates retesting the entire application to ensure a quality release. Manual testing can no longer keep pace in this dynamic development environment. WinRunner helps you automate the testing process, from test development to execution. You create adaptable and reusable test scripts that challenge the functionality of your application. Prior to a software release, you can run these tests in a single overnight run—enabling you to detect defects and ensure superior software quality.

WinRunner Testing Modes

WinRunner facilitates easy test creation by recording how you work on your application. As you point and click GUI (Graphical User Interface) objects in your application, WinRunner generates a test script in the C-like Test Script Language (TSL). You can further enhance your test scripts with manual programming. WinRunner includes the Function Generator, which helps you quickly and easily add functions to your recorded tests.

The WinRunner Testing Process



Create the GUI Map

The first stage is to create the GUI map so WinRunner can recognize the GUI objects in the application being tested. Use the RapidTest Script wizard to review the user interface of your application and systematically add descriptions of every GUI object to the GUI map. Alternatively, you can add descriptions of individual objects to the GUI map by clicking objects while recording a test.

Note that when you work in *GUI Map per Test* mode, you can skip this step. For additional information, see Chapter 3, “Understanding How WinRunner Identifies GUI Objects.”

Create Tests

Next, you create test scripts by recording, programming, or a combination of both. While recording tests, insert checkpoints where you want to check the response of the application

being tested. You can insert checkpoints that check GUI objects, bitmaps, and databases. During this process, WinRunner captures data and saves it as *expected results*—the expected response of the application being tested.

Debug Tests

You run tests in Debug mode to make sure they run smoothly. You can set breakpoints, monitor variables, and control how tests are run to identify and isolate defects. Test results are saved in the debug folder, which you can discard once you’ve finished debugging the test.

When WinRunner runs a test, it checks each script line for basic syntax errors, like incorrect syntax or missing elements in **If**, **While**, **Switch**, and **For** statements. You can use the **Syntax Check** options (**Tools >Syntax Check**) to check for these types of syntax errors before running your test.

Run Tests

You run tests in Verify mode to test your application. Each time WinRunner encounters a checkpoint in the test script, it compares the current data of the application being tested to the expected data captured earlier. If any mismatches are found, WinRunner captures them as *actual results*.

View Results

You determine the success or failure of the tests. Following each test run, WinRunner displays the results in a report. The report details all the major events that occurred during the run, such as checkpoints, error messages, system messages, or user messages.

If mismatches are detected at checkpoints during the test run, you can view the expected results and the actual results from the Test Results window. In cases of bitmap mismatches, you can also view a bitmap that displays only the difference between the expected and actual results.

You can view your results in the standard WinRunner report view or in the Unified report view. The WinRunner report view displays the test results in a Windows-style viewer. The Unified report view displays the results in an HTML-style viewer (identical to the style used for QuickTest Professional test results).

Report Defects

If a test run fails due to a defect in the application being tested, you can report information about the defect directly from the Test Results window.

This information is sent via e-mail to the quality assurance manager, who tracks the defect until it is fixed.

You can also insert **tddb_add_defect** statements to your test script that instruct WinRunner to add a defect to a TestDirector project based on conditions you define in your test script.

WinRunner at a Glance

This chapter explains how to start WinRunner and introduces the WinRunner window.

This chapter describes:

Starting WinRunner

The Main WinRunner Window

The Test Window

Using WinRunner Commands

Loading WinRunner Add-Ins

Starting WinRunner

To start WinRunner:

Choose **Programs > WinRunner > WinRunner** on the **Start** menu.

The WinRunner Record/Run Engine icon appears in the status area of the Windows taskbar. This engine establishes and maintains the connection between WinRunner and the application being tested.

11. WinRunner User's Guide • Starting the Testing Process

The WinRunner Add-in Manager dialog box opens.



Note: The first time you start WinRunner, “What’s New in WinRunner” help also opens.

The WinRunner Add-in Manager dialog box contains a list of the add-ins available on your computer. The WinRunner installation includes the ActiveX Controls, PowerBuilder, Visual Basic, and WebTest add-ins.

You can also extend WinRunner's functionality to support a large number of development environments by purchasing external WinRunner add-ins. If you install external WinRunner add-ins they are displayed in the Add-in Manager together with the core add-ins. When you install external add-ins, you must also install a special WinRunner add-in license.

12. Chapter 2 • WinRunner at a Glance

The first time you open WinRunner after installing an external add-in, the Add-in Manager displays the add-in, but the check box is disabled and the add-in name is grayed. Click the **Add-in License** button to install the Add-in license.

Select the add-ins you want to load for the current session of WinRunner. If you do not make a change in the Add-in Manager dialog box within a certain amount of time, the window closes and the selected add-ins are automatically loaded. The progress bar displays how much time is left before the window closes.

The Welcome to WinRunner window opens. From the Welcome to WinRunner window you can click **New Test** to create a new test, click **Open Test** to open an existing test, or click **Quick Preview** to view an overview of WinRunner in your default browser.

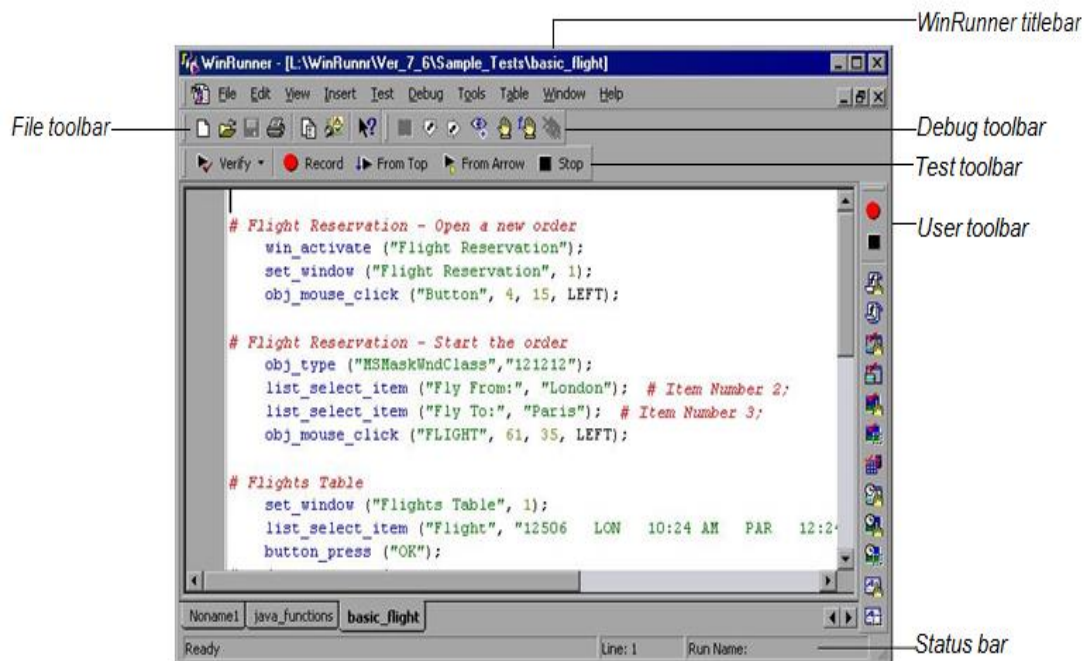


The Main WinRunner Window

The main WinRunner window contains the following key elements:

1. WinRunner title bar—displays the name and path of the currently open test.
2. File toolbar—provides easy access to frequently performed tasks, such as opening and saving tests, and viewing test results.
3. Debug toolbar—provides easy access to buttons used while debugging tests.
4. Test toolbar—provides easy access to buttons used while running and maintaining tests.
5. User toolbar—displays the tools you frequently use to create test scripts. By default, the User toolbar is hidden. To display the Debug toolbar, choose **View > User Toolbar**.

Status bar—displays information on the current command, the line number of the insertion point, and the name of the current results folder



6.b. How win runner identifies GUI objects in an application and describes the two modes for organizing GUI map files.

What is Gui Map?

“GUI Map” provides a layer of indirection between the objects described in the script and the widgets created by the application. The GUI Map is made up of all currently loaded GUI Map files. GUI Map files are viewed in the GUI Map Editor. The GUI map file contains the logical names and physical descriptions of GUI objects. WinRunner stores information it learns about a window or object in a GUI Map. The GUI map provides a centralized object repository, allowing testers to verify and modify any tested object. These changes are then automatically propagated to all appropriate scripts, eliminating the need to build new scripts each time the application is modified. When WinRunner runs a test, it uses the GUI map to locate objects. It reads an object’s description in the GUI map and then looks for an object with the same properties in the application being tested. Each of these objects in the GUI Map file will be having a logical name and a physical description. When WinRunner learns the description of a GUI object, it does not learn all its properties. Instead, it learns the minimum number of properties to provide a unique identification of the object. Each object is identified within the scope of its parent window, not the entire application.

An Example of how WinRunner uses a logical name and physical description to identify an object:

“Print” for a Print dialog box, or “OK” for an OK button. This short name connects WinRunner to the object’s longer physical description.

Logical Name:

This is the name that appears in the test script when user records an application. Usually WinRunner uses attached text that WinRunner can read as the Logical name. WinRunner checks that there are no other objects in the GUI map with the same name.

```
set_window ("Readme.doc - WordPad", 10);  
menu_select_item ("File; Print... Ctrl+P");  
set_window ("Print", 12);  
button_press ("OK");  
Physical Description:
```

The physical description contains a list of the object’s physical Properties. The Print dialog box, for example, is identified as a window with the label “Print”


```
. Readme.doc window: {class: window, label: "Readme.doc - WordPad"}  
File menu: {class: menu_item, label: File, parent: None}  
Print command: {class: menu_item, label: "Print... Ctrl+P", parent: File}  
Print window: {class: window, label: Print}  
OK button: {class: push_button, label: OK}
```

For each class, WinRunner learns a set of default properties. Each default property is classified “obligatory” or “optional”. An obligatory property is always learned (if it exists).

An optional property is used only if the obligatory properties do not provide unique identification of an object. These optional properties are stored in a list. WinRunner selects the minimum number of properties from this list that are necessary to identify the object. It begins with the first property in the list, and continues, if necessary, to add properties to the description until it obtains unique identification for the object. In cases where the obligatory and optional properties do not uniquely identify an object, WinRunner uses a selector to differentiate between them. Two types of selectors are available:

Location selector:-The location selector uses the spatial order of objects within the window, from the top left to the bottom right corners, to differentiate among objects with the same description.

Index selector:-The index selector uses numbers assigned at the time of creation of objects to identify the object in a window. Use this selector if the location of objects with the same description may change within a window.

Consider an example where a form has two OK buttons

Script:-

```
set_window ("Form1", 2);  
button_press ("OK") ;{ class: push_button, label: OK, MSW_id: 1}  
button_press ("OK_1") ;{ class: push_button, label: OK, MSW_id: 2}
```

WinRunner recorded the object logical names as OK and OK_1, in Physical description, both the buttons are having the same class and label properties. So, WinRunner assigned the third property to both the buttons "MSW_id (Microsoft windows Id) which is assigned by operating system. When we run the script, WinRunner will recognize those objects by MSW_id as the id is different for both the OK buttons. User can modify the logical name or the physical description of an object in a GUI map file using the GUI Map Editor. Changing the logical name of an object is useful when the assigned logical name is not sufficiently descriptive or is too long. Changing the physical description is necessary when the property value of

an object changes.

Using the GUI Spy, user can view the properties of any GUI object on users desktop. User use the Spy pointer to point to an object, and the GUI Spy displays the properties and their values in the GUI Spy dialog box. User can choose to view all the properties of an object, or only the selected set of properties that WinRunner learns.

There are two modes for organizing GUI map files.

Global GUI Map file: - a single GUI Map file for the entire application

GUI Map File per Test: - WinRunner automatically creates a GUI Map file for each test created

Global GUI Map File is the default mode. As the name suggests in Global mode a single script is created for the entire application.

Using Rapid Test Script Wizard option we can learn the entire application but this option is not available in the GUI File Test mode.

In Global GUI Map file, it learns all the objects in the window into one temporary GUI Map file. We can see this temporary file by Tools -> GUI Map Editor -> L0 . In this file all objects and their property values will be stored. We need to save this GUI Map file explicitly. If it's not saved, then all the entries will remain in the temporary file. We can specify whether we have to load this temporary GUI Map file should be loaded each time in the General Options.

In GUI Map File Per Test a script is generated for each and every window/ screen in the application. In GUI Map File Per Test if we save the test script it implicitly saves all the GUI objects in a separate GUI file.

If an object isn't found in the GUI Map during recording, WinRunner reads its attributes and adds it to the Temporary GUI Map file. During playback, it doesn't matter which GUI Map file defines an object. Objects may be identified from any loaded GUI Map file whether it is temporary file or GUI Map file. If the GUI Map already contains an object, another file with that object cannot be loaded into the GUI Map. GUI files with unsaved changes preceded by asterisk (*).Temporary GUI always loads automatically.

WinRunner fails to identify an object in a GUI due to various reasons.

- i. The object is not a standard windows object.
- ii. If the browser used is not compatible with the WinRunner version, GUI Map Editor will not be able to learn any of the objects displayed in the browser window

The GUI Map Editor displays the various GUI Map files created and the windows and objects learned in to them with their logical name and physical description. We can invoke GUI Map Editor from the Tools Menu in WinRunner. The different options available in the GUI Map Editor are

Learn: -Enables users to learn an individual GUI object, a window, or the entire GUI objects within a window.

Modify:-Opens the Modify dialog box and allows user to edit the logical name and the physical description of the selected GUI object.

Add: -adds a GUI objects to the open GUI Map files.

Delete: -deletes the selected GUI objects from the open GUI Map files.

Copy (Expanded view only): -copies the selected GUI objects to the other GUI map file in the GUI Map Editor.

Move (Expanded view only): -moves the selected GUI objects to the other GUI map file in the GUI Map Editor.

Show: -highlights the selected GUI object if the object is visible on the screen.

Find: -Helps users to easily locate a specific GUI object in the GUI map.

Expand (GUI Files view only): -expands the GUI Map Editor Dialog box, enabling the user to copy or move GUI objects between open GUI Map files.

Collapse (GUI Files view only): -collapses the GUI Map Editor Dialog box.

Trace (GUI Files view only): -Enables user to trace a GUI object that appears in more than one GUI Map file.

User can clear a GUI Map file using the “Clear All” option in the GUI Map Editor.

When the user is working with GUI map per test mode and if the user clear the temporary GUI map, the GUI map test information will not be saved with the test and the test may fail.

Filters Options

GUI Map Editor has a Filter option which enables user to define which GUI objects to display in the GUI Map Editor, there are 3 options. Filter by Logical Name:-If selected, displays only those GUI objects whose Logical Names contain the substring user specified.

Filter by Physical Description:-If selected, displays only those GUI objects whose Physical Descriptions contain the substring user specified.

Filter by Class:-If selected, displays only those GUI objects in the class user specified.

Saving Changes to the GUI Map

When the user makes some modification to the physical description or logical name within a GUI map file then the user must save the changes before ending the testing session and exiting WinRunner. User need not save the changes manually if the user is working in the GUI Map File per Test mode. Changes are saved automatically with the test. If the user adds new windows from a loaded GUI map file to the temporary GUI map file, then when the user save the temporary GUI map file, the New Windows dialog box opens. Prompting to add the new windows to the loaded GUI map file or save them in a new GUI map file.

User can load GUI map in two ways

a) Using function

GUI_load (file_name);

file_name The full path of the GUI map.

If the user is not specifying a full path, then WinRunner searches for the GUI map relative to the current file system directory. So, the user must always specify a full path to ensure that WinRunner will find the GUI map.

b) Using map editor

From GUI files drop down in the Map editor the user can select the file name. When the user selects a file name then the GUI file will be loaded.

Unload GUI map files.

GUI_close();

To unload a specific GUI Map file

GUI_close_all;

To unload all the GUI Map files loaded in the memory.

While working in the GUI Map File per Test mode, WinRunner automatically creates, saves, and loads a GUI map file with each test user create. When user work in the Global GUI Map File mode it enables user to save information about the GUI of the application in a GUI map that can be referenced by several tests. When the application changes instead of updating each test individually, user can merely update the GUI map that is referenced by the entire group of tests. The GUI Map File Merge Tool enables user to merge multiple GUI map files into a single GUI map file. Before merging GUI map files, user must specify at least two source GUI map files to merge and at least one GUI map file as a target file. The target GUI map file can be an existing file or a new (empty) file.

Auto merge: - The merge tool merges all GUI map files, and prompts user only if there are conflicts to resolve between the files. Manual merge:-user merges each GUI map file manually. The merge tool prevents user from creating conflicts while merging the files.

Many applications contain custom GUI objects. A custom GUI object is any object not belonging to one of the standard classes used by WinRunner these objects are therefore assigned to the generic “object” class. When WinRunner records an operation on a custom object, it generates obj_mouse_click statements in the test script. If a custom object is similar to a standard object, user can map it to one of the standard classes. user can also configure the properties WinRunner uses to identify a custom object during Context Sensitive testing. The mapping and the configuration user set are valid only for the current WinRunner session. To make the mapping and the configuration permanent, user must add configuration statements at the startup test script.

A startup test is a test script that is automatically run each time when the user start WinRunner. User can create startup tests that load GUI map files and compiled modules, configure recording, and start the application under test. User can designate a test as a startup test by entering its location in the Startup Test box in the Environment tab in the General Options dialog box. User can use the RapidTest Script wizard to create a basic startup test called myinit that loads a GUI map file and the application being tested. When working in the GUI Map File per Test mode the myinit test does not load GUI map files.

Sample Startup Test

Start the Flight application if it is not already displayed on the screen.
#invoke_application statement, which starts the application being tested.

```
if ((rc=win_exists("Flight")) == E_NOT_FOUND)
invoke_application("w:\\flight_app\\flight.exe", "", "w:\\flight_app",SW_SHOW);
```

Load the compiled module "qa_funcs". load statements, which load compiled modules #containing user-defined functions that users frequently call from their test scripts.

```
load ("qa_funcs", 1, 1);
```

Load the GUI map file "flight.gui". GUI_load statements, which load one or more GUI #map files. This ensures that WinRunner recognizes the GUI objects in the application #when the user run tests.

```
GUI_load ("w:\\qa\\gui\\flight.gui");
```

```
# Map the custom "borbtn" class to the standard "push_button" class. set_class_map  
statement configure how WinRunner records GUI objects in application.  
set_class_map ("borbtn", "push_button");
```

Deleting a Custom Class

User can delete only custom object classes. The standard classes used by WinRunner cannot be deleted.

Virtual object

Applications may contain bitmaps that look and behave like GUI objects. WinRunner records operations on these bitmaps using `win_mouse_click` statements. By defining a bitmap as a virtual object, user can instruct WinRunner to treat it like a GUI object such as a virtual push buttons, radio buttons, check buttons, lists, or tables when the user record and run tests. If none of these is suitable, user can map a virtual object to the general object class.

WinRunner identifies a virtual object according to its size and its position within a window, the x, y, width, and height properties are always found in a virtual object's physical description. If these properties are changed or deleted, WinRunner cannot recognize the virtual object. If the user move or resize an object, user must use the wizard to create a new virtual object. The virtual object should not overlap GUI objects in application (except for those belonging to the generic "object" class, or to a class configured to be recorded as "object"). If a virtual object overlaps a GUI object, WinRunner may not record or execute tests properly on the GUI object.

Advantages of GUI Map

Maintainability

If a button label changes in the application, update the button description once in the GUI map rather than in 500 tests

Readability

```
button_press ("Insert")
```

instead of

```
button_press("{class: ThunderSSCommand}");
```

Portability

Use the same script for all platforms, with a different GUI map for each platform

6.C. How to record a test script and explain the basics of test script language

What is a test script?

A **test script** in software **testing** is a set of instructions that will be performed on the system under **test** to **test** that the system functions as expected. There are various means for executing **test scripts**. **Manual testing**. These are more commonly called **test cases**.

How do I create a test script?

Procedure

1. Open a test case.
2. From the Test Scripts section of the test case, click the Create Test Script icon ().
3. In the New Test Script dialog box, in the Name field, type a descriptive name that identifies the purpose of the script.
4. Optional: Type a description.
5. The type is Manual by default.
6. Click OK.

Test Script Language (TSL) is a scripting language with syntax similar to C language. There are 4 categories of TSL functions. Each category of functions is for performing specific tasks: These categories are as follows:

- **Analog functions:** These functions are used when you record in Analog mode, a mode in which the exact coordinates of the GUI map are required.
- **Context sensitive functions:** These functions are used where the exact coordinates are not required.
- **Customization functions:** These functions allow the user to improve the testing tool by adding functions to the Function Generator.
- **Standard functions:** These functions include all the basic elements of programming language like control flow statements, mathematical functions, string related functions etc.

The most frequently used functions in each category are as follows:

- **click:** To click a mouse button.
Syntax: click(mouse_button,[time])
mouse_button - specify LEFT, RIGHT or MIDDLE
- **dbl_click:** To double click a mouse button.
Syntax: dbl_click(mouse_button,[time])
mouse_button - specify LEFT, RIGHT or MIDDLE
- **get_x:** To get the current x coordinate of the mouse pointer.
Syntax: get_x();
Return value: integer

- **get_y:** To get the current y coordinate of the mouse pointer.
Syntax: get_y();
Return value: integer
- **waitjwindow:** Waits for a window bitmap to appear for synchronizing the test procedure.
Syntax: wait_window(time, image, window, width, height, x, y [, rebel, relyl, relx2, rely2])
- **get_text:** To read the text from the screen specified by the location.
Syntax: get_text(location);
The location can be xl>yl, x2, y2 or x ,y or ()(no location). When no location is specified, it considers the point closest to the mouse pointer.
- **button_check_info:** To check the value of the button property.
Syntax: button_check_info (Button, property, property_value);
Button - Name of the button
Property - The property to be checked
Property_value - The property value
- **button_check_state:** To check the state of the radio button or check box.
Syntax: button_check_state (button, state);
Button - Name of the button.
State - The state of the button (1 for ON and 0 for OFF)
- **button_get_info:** Returns the value of a button property.
Syntax: button_get_info (button, property, out_value);
Button - Name of the button.
Property - The property of the button to be retrieved.
Out_value - The variable where the retrieved property value will be stored.
- **button_get_state:** To get the state of a radio button or check button.
Syntax: button_get_state (Button, Out_state);
Button - The name of the radio button or check button.
Out_state - The variable where the retrieved state of the button will be stored.
- **button_press:** clicks on a push button. Syntax: button press (Button);
Button - The logical name of the button.
- **button set:** To set the state of a radio or check button.
Syntax: button_set (Button, State);
Button - The name of the button.
State - The state of the button,
The state of the button can either be ON, OFF, DIMMED or TOGGLE (toggles between ON and OFF).
- **button_wait_info:** Waits for the value of a button property.
Syntax: button_wait_info (Button, Property, Value, time);
Button - The name of the button.
Property - The property of the button

Value - The value of the property

Time - The time interval before the next statement to be executed.

- **ddtjdose:** To close the data table file. Syntax: `ddt_close(data_table_name);`

Data_table_name - name of the data table.

- **ddt_close_all_tables:** To close all open tables. Syntax:

`ddt_close_all_tables();`

- **db_disconnect:** To disconnect from the database and end the database session.

Syntax: `db_disconnect (Session_name);`

Session_name - The logical name of the database.

ddt_get_row_count: To retrieve the number of rows in a data table.

Syntax: `ddt_get_row_count (Data_table_name, out_rows_count);`

Data_table_name - The name of the data table.

out_rows_count - The variable in which the row count is stored.

- **ddt_get_current_row:** To retrieve the current row from the data table.

Syntax: `ddt_get_current_row (Data_table_name, out_row);`

Data_table_name - The name of the data table.

Out_row - The variable that stores the output row from the data table.

- **ddt_next_row:** To point to the next row in the data table.

Syntax: `ddt_next_row (Data_table_name);`

Data_table_name - The name of the data table.

- **ddt_open:** To create or open a data table file so that WinRunner can access it.

Syntax: `ddt_open (Data_table_name[, Mode]);`

Data_table-name - The name of the Data table.

Mode - The mode in which the table has to be opened.

The various modes are: DDT_MODE_READ (read-only) or

DDT_MODE_READWRITE (read or write).

- **edit_check_info:** To check the value of an edit object property.

Syntax: `edit_check_info (Edit, Property, Property_value);`

Edit - The logical name of the edit object.

Property - The property to be checked.

Property_value - The values of the property.

- **edit_delete:** To delete the contents of an edit object.

Syntax: `edit_delete (Edit, Start_column, End_column);`

Edit - The logical name of the edit object.

Start_column - The column where the text starts.

End_column - The column where the text ends.

- **obj__check_gui:** To compare the current GUI object data to expected data.

Syntax: `obj_check_gui (Object, Checklist, Expected_results_file, Time);`

Object - The logical name of the GUI object.

Checklist - The name of the checklist defining the GUI checks.

Expected_results_file - The name of the file that stores the expected GUI data.

Time - Specifies the delay between the previous input and the next input.

- **win_check_gui:** To compare the current GUI data to expected GUI data for a window.

Syntax: win_check_gui (Window, Checklist, Expected_results_file, Time);

Object - The logical name of the GUI object.

Checklist - The name of the checklist defining the GUI checks.

Expected_results_file - The name of the file that stores the expected GUI data.

Time - Specifies the delay between the previous input and the next input.

- **list_activate_item:** To activate an item in a list.

Syntax: list_activate_item (List, Item [, Offset]);

List - The logical name of the list.

Item - The item to be activated in the list.

Offset - The horizontal offset of the click location in pixels.

- **menu_get_info:** To return the value of a menu property.

Syntax: menu_get_info (Menu, Property, Out_value);

Menu - The logical name of the menu.

Property - The property to be checked.

Out_value - The variable in which the value of the specified property is stored.

- **menu_get__item:** To return the contents of a menu item.

Syntax: menu_get_item (Menu, Item_number, Out_contents);

Menu - The logical name of the menu.

Item_number - The numerical position of the item in the menu.

Out_contents - The output variable to which the value of the designated menu item is assigned.

- **obj_check_bitmap:** To compare an object bitmap to an expected bitmap.

Syntax: obj_check_bitmap (Object, Bitmap, Time [, x, y, width, height]);

Object - The logical name of the GUI object.

Bitmap - A string expression that identifies the captured bitmap.

Time - The time interval for which the application waits for the expected bitmap

- **obj_check_gui:** To compare the current GUI object data to expected data.

Syntax: obj_check_gui (Object, Checklist, Expected_results_file, Time);

Object - The logical name of the GUI object

Checklist - The name of the checklist defining the GUI checks.

Expected_results_file - The name of the file that stores the expected GUI

6.d. how to synchronize a test when the application responds slowly.

In WinRunner, it takes by default one second to execute the next statement. But sometimes there may be a case where the WinRunner has to wait for a few seconds to accept the data from the user or wait till the current operation is completed, before executing the next statement.

The Synchronization is required in the following cases:

- When data has to be retrieved from the database.
- When a progress bar has to reach 100%.
- To wait till some message appears.

Though, by default, WinRunner takes one second to execute the next statement, it is possible to change the default time to any desired value. Changing the value of the "Timeout for Checkpoints and CS Statement" option in the Run tab of the General option's dialog can do this. Settings -> General Options.

In such a case, it will affect the entire application and as a result the entire process of testing becomes very slow. To avoid this, we find out the statement in the test script where the problem may occur and create a synchronization point. The synchronization point tells WinRunner to wait for specified interval for the specified response.

Creating the synchronization point is explained here with the example of a database application. The EmpDB application discussed here is a Visual Basic application. This application is used to create, open, modify, and delete the employee details. The procedure for installing this sample application is given in

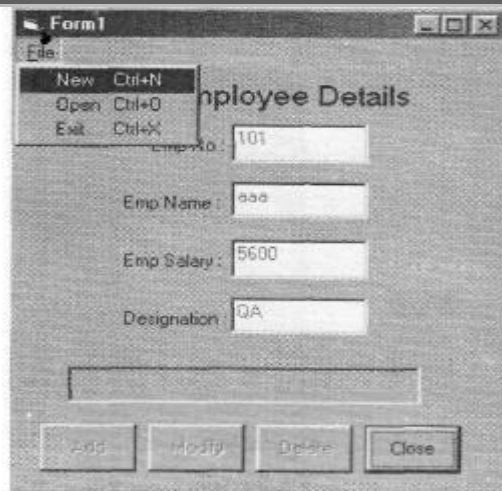
Appendix E. The first screen of EmpDB is shown in Figure.

Creating the Test Case

Step 1: Open WinRunner application and create a new test case. Step 2: Open the EmpDB application.

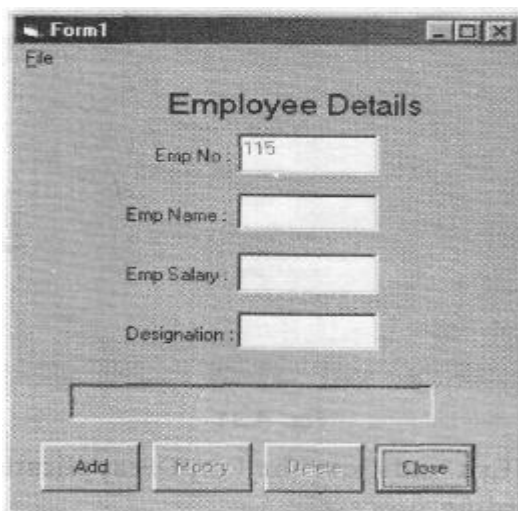
Step 3: Start recording the test case in Context Sensitive mode Create -> Record - Context Sensitive or click the button from WinRunner toolbar.

Step 4: Create a new Employee record as shown in Figure.



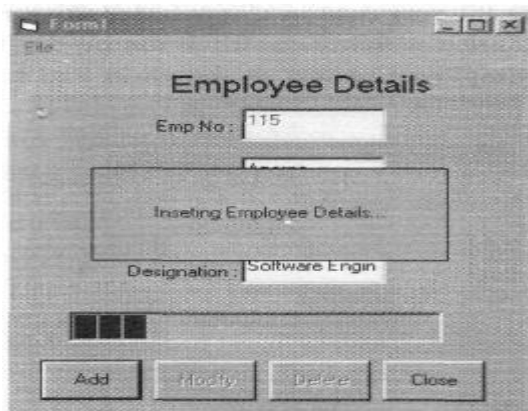
The screenshot shows a window titled 'Form1' with a menu bar containing 'File'. The 'File' menu is open, showing options: 'New' (Ctrl+N), 'Open' (Ctrl+O), and 'Exit' (Ctrl+X). The main area is titled 'Employee Details' and contains four input fields: 'Emp No.' with the value '101', 'Emp Name' with the value 'aaa', 'Emp Salary' with the value '5600', and 'Designation' with the value 'QA'. Below these fields is a horizontal line, and at the bottom are four buttons: 'Add', 'Modify', 'Delete', and 'Close'.

Step 5: When you click "New" menu option, a dialog is displayed where the Emp No. is automatically incremented and prompts you to enter the remaining details such as Emp Name, Emp Salary, Designation as shown in Figure.



The screenshot shows the 'Employee Details' form with the 'Emp No.' field now containing '115'. The 'Emp Name', 'Emp Salary', and 'Designation' fields are empty. The 'Add', 'Modify', 'Delete', and 'Close' buttons are still at the bottom.

Step 6: After entering all the details click on "Add" button to insert the record into the database as shown in Figure.



The screenshot shows the 'Employee Details' form with the 'Emp No.' field containing '115' and the 'Designation' field containing 'Software Engin'. A message box is displayed in the center of the form with the text 'Inserting Employee Details...'. The 'Add', 'Modify', 'Delete', and 'Close' buttons are at the bottom.

Step 7: When the insertion is completed, "Insert Done" message appears in the status bar as shown in Figure.

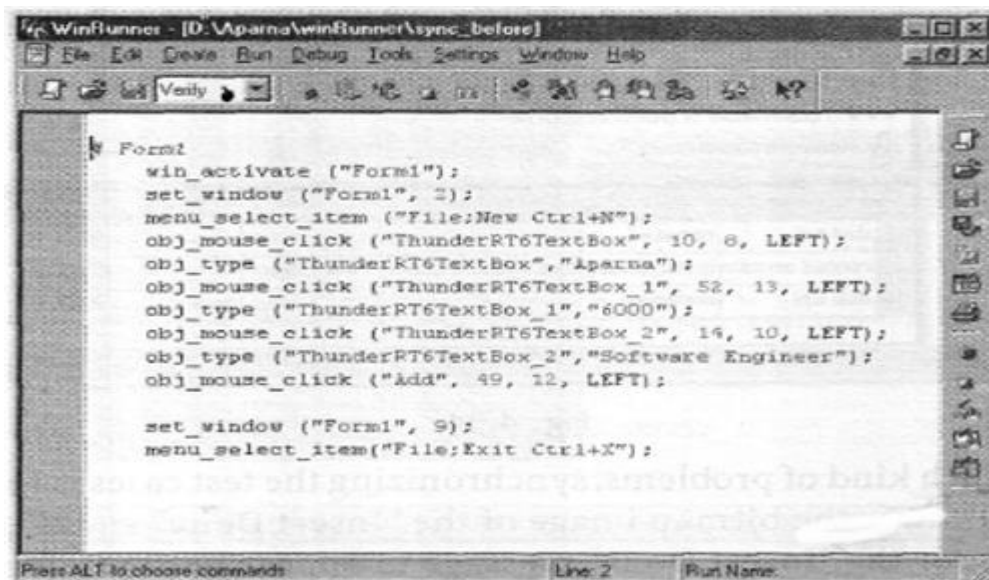


Step 8: Select File -> Exit and close the EmpDB application.

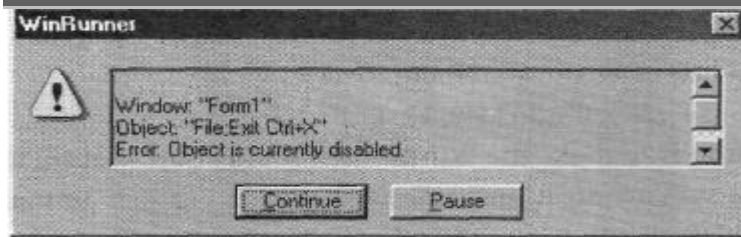
Step 9: Stop the recording process.

Create-^ Stop Recording or click the button on the WinRunnertoolbar

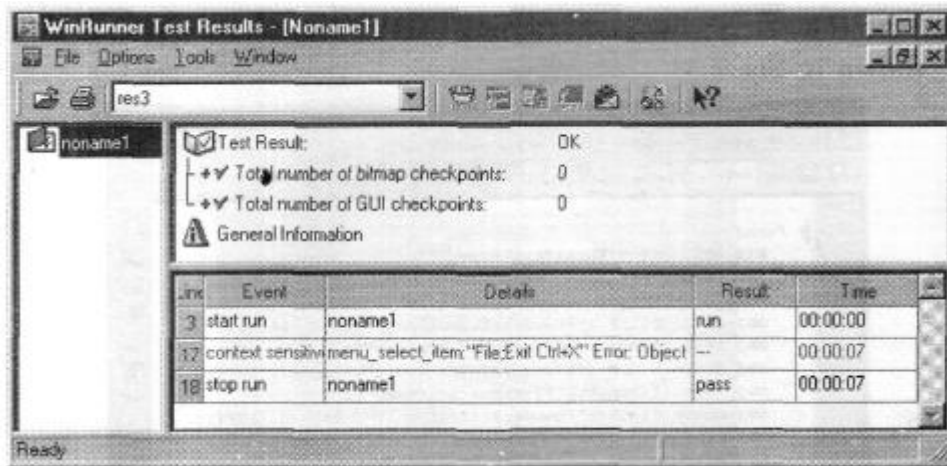
Step10: Save the test script with name "sync_before". The test script is displayed as shown in Figure.



Step 11: Run the test case by selecting "Run from Top" option from the Run menu or Click button on WinRunner toolbar. An error message, as shown in Figure appears because WinRunner fails to select the menu option "File - Exit". The error occurred because WinRunner did not wait until the insertion action is completed. Click "Pause" button.



If you click on the "Continue" button, it executes the remaining lines in the test script and displays the error report as shown in Figure



To avoid such kind of problems, synchronizing the test cases is required. WinRunner application captures the bitmap image of the "Insert Done" and when you run the test script, it waits for the "Insert Done" message to appear in the status bar.

Synchronizing the Test Cases

We will now create a synchronization point into the test script..

Step I: Open the test script if it is closed.

File -> Open and select the "sync_be-before" file.

Step 2: Place the cursor at the point where the test has to be synchronized. Insert a line after the statement

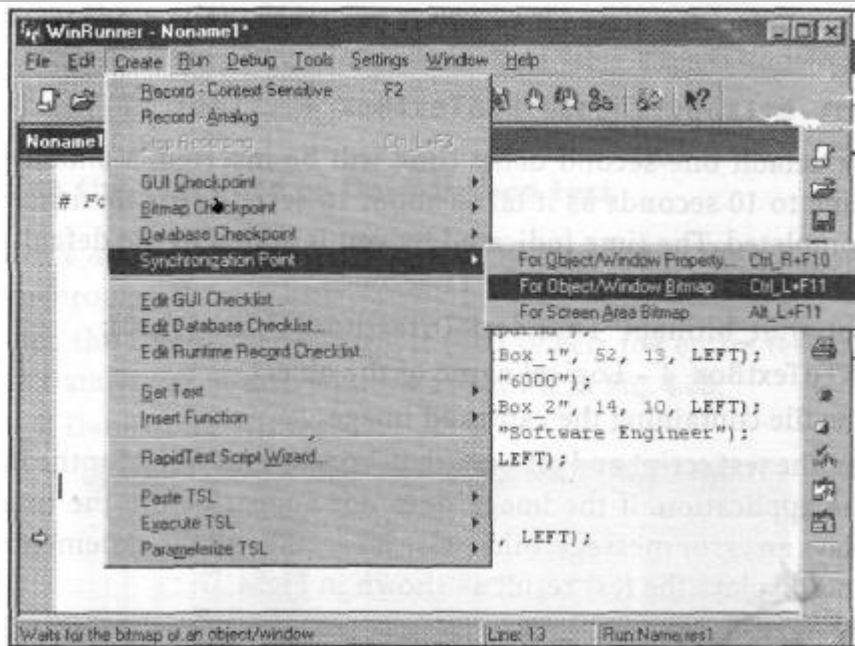
`obj_mouse_click ("Add", 49,12, LEFT);`

as our aim is to make the WinRunner application wait till the insertion is over.

Step 3: Insert a synchronization point as shown in Figure to make WinRunner wait

until the insertion is completed. Create -> Synchronization Point -> For Object/Window Bitmap

Once you select the status message, it inserts the following statement into the test script: `obj_wait_bitmap("ThunderRT6TextBox_4"."Irngl",1);`



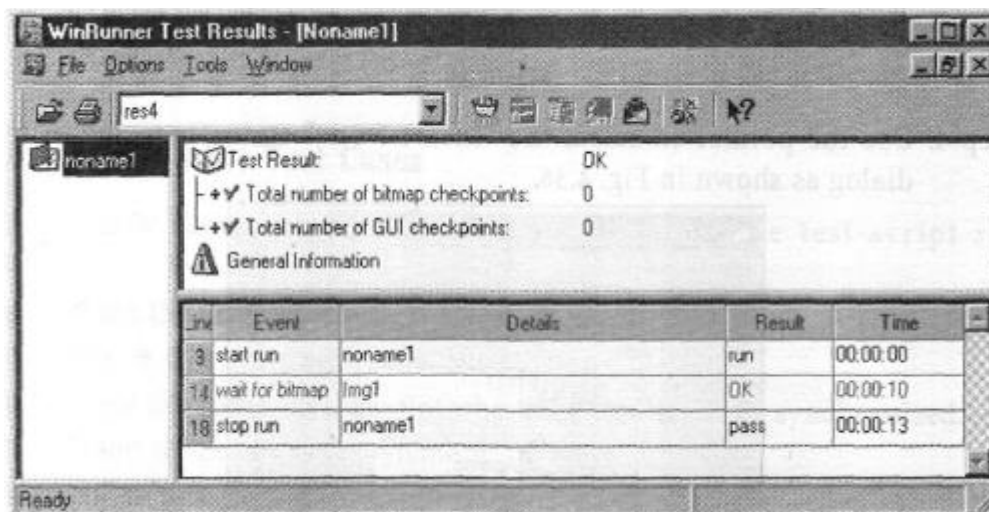
Step 5: By default one-second delay time will be inserted. Manually change the wait time to 10 seconds as it takes about 10 seconds for the insertion action to be completed. The time indicated by you is added to the default time interval. So totally WinRunner waits for 11 seconds.

```
obj_wait_bitmap("ThunderRT6TextBox_4","Tmg1",10);
```

ThunderRT6TextBox_4 - Logical name of the object.

Img1 - The file containing the captured image

Step 6: Run the test script and observe that, WinRunner waits for the image to appear in the application. If the image does not appear before the timeout time, it displays an error message; otherwise it executes all the statements in the test script and displays the test result as shown in Figure



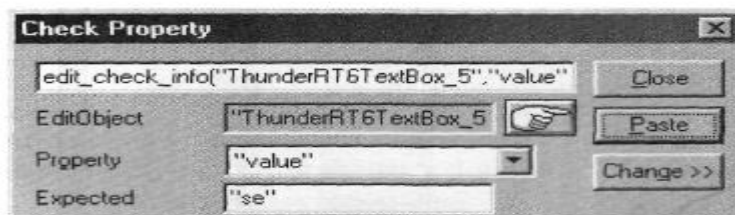
6. E. how to create test that checks GUI objects and compare the behavior of GUI objects in different versions of the sample application

It is possible to check the behavior of the objects in the application by creating the GUI Checkpoints. The GUI Checkpoints help to find the defects in the application, by examining the objects.

Checking a Single property

In order to check a single property of one particular object (for example, if you want to check whether a button is enabled or disabled), you need to create the checkpoint for a single object as follows: Go to CreateGUI Checkpoint -> For Single Property

On selecting this option, it prompts you to select the object for which the checkpoint has to be created. Select the required object. It then displays the "Check property" dialog as shown in Figure. Select the required property, on which the object has to be checked, by selecting a value from the "Property" combo box. WinRunner automatically assigns the default values to that function.



For example, if you want to create the checkpoint for an edit control where you want to check that except for one value i.e. "se", if you enter any other value, it should generate error. In such cases, select the Property as "value" and in the Expected field enter the string "se" as shown in Figure. When you run the test script, the test fails if you enter any value other than "se".

Checking a Single Object

If you want to create checkpoint for multiple properties of a single object, then you need to follow the procedure given next. Let us consider the EmpDB application to create checkpoint for multiple properties of a single object. Consider the case where you want to create a checkpoint for the "Designation" field in the EmpDB application. If you enter any other designation except for "Software Engineer", it should display error. To create a checkpoint on the "Designation" field follow the steps below.

Step 1: Create a new test.

File -> New

Step 2: Open the EmpDB application

Step 3: Start Recording in Context Sensitive mode.

Create -> Record - Context Sensitive

Step 4: Insert a record into EmpDB

1. File->New
2. Enter all the fields
3. Click "Add" to insert the record

Step 5: Synchronize the test (Refer to the section Synchronization of test cases for details)

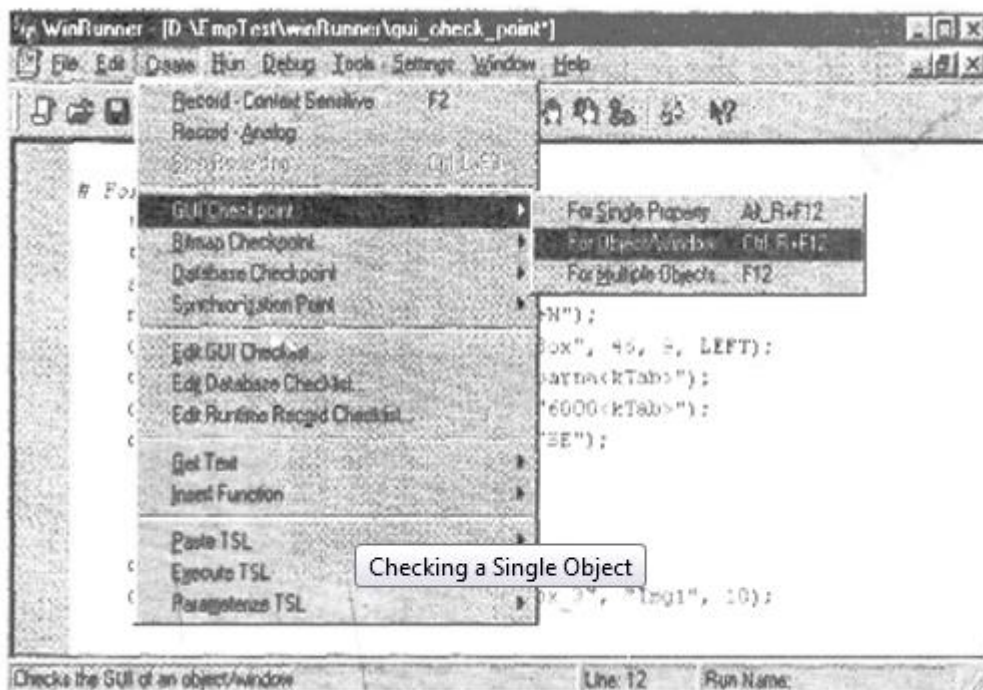
Step 6: Stop Recording.

Create -> Stop Recording

Step 7: Place the cursor where the "Designation" field has to be validated.

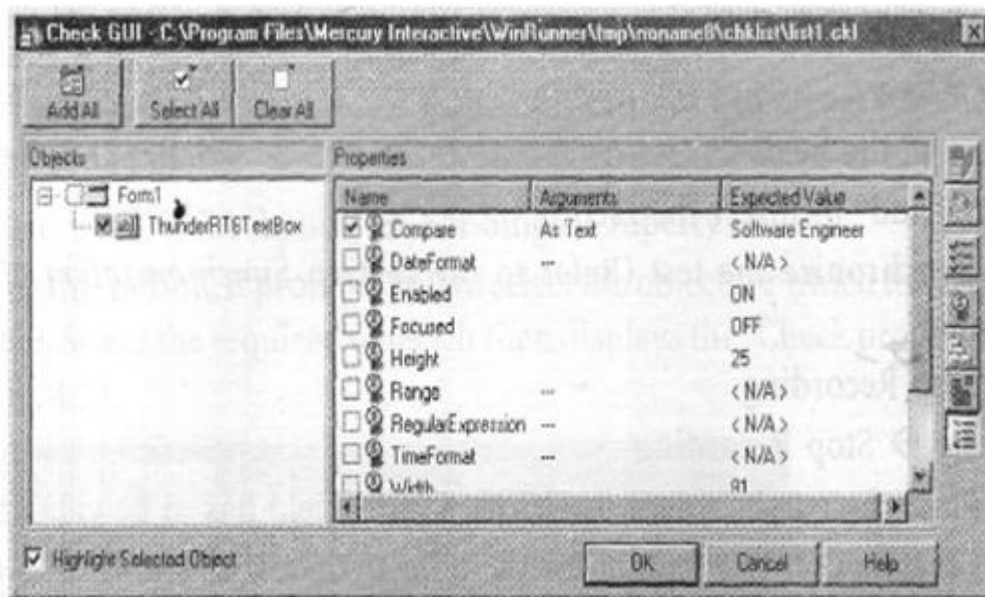
Step 8: Create a GUI Checkpoint for the object/window as shown in Figure.

Create -> GUI Checkpoint -> For Object/Window



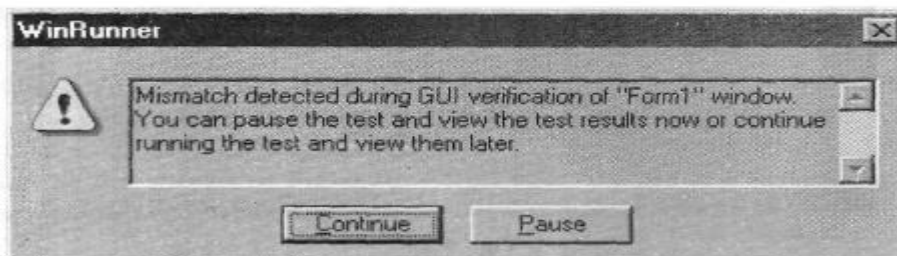
step 9: Double-click on the object on which the checkpoint has to be created.

Step 10: When you double click on the object, the "Check GUI" dialog appears which displays all the available properties of the control. Since we want to compare the text of the control, double click on the "Expected value" of "Compare" property. This will display "Edit Expected value" dialog prompting you to enter the expected value. Enter "Software Engineer" as shown in Figure. Change the other properties on which checkpoints have to be created, if required.



Step 11: This will insert the following statement in the test script where the cursor is pointing. `objcheckgui("ThunderRT6TextBox","listl.ckl","gui1", 1);`

Step 12: If you enter the Designation as "SE" instead of "Software Engineer" and run the test script, it displays the message as shown in Figure.



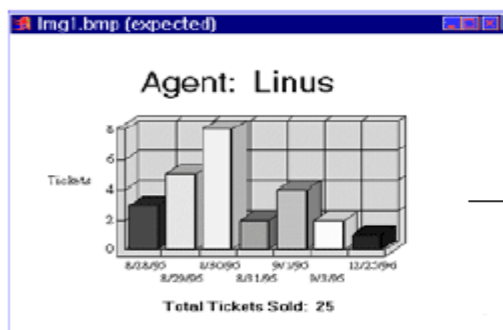
6.F. How to create and run a test that checks bitmaps in your application and run the test on different versions of the sample application and examine any differences,

pixel by pixel.

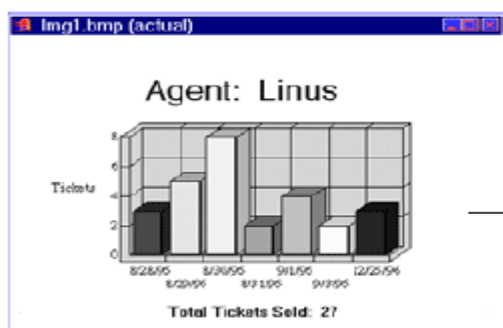
About checking bitmaps

You can check an object, a window, or an area of a screen in your application as a bitmap. While creating a test, you indicate what you want to check. WinRunner captures the specified bitmap, stores it in the expected results folder (*exp*) of the test, and inserts a checkpoint in the test script. When you run the test, WinRunner compares the bitmap currently displayed in the application being tested with the *expected* bitmap stored earlier. In the event of a mismatch, WinRunner captures the current *actual* bitmap and generates a *difference* bitmap. By comparing the three bitmaps (expected, actual, and difference), you can identify the nature of the discrepancy.

Suppose, for example, your application includes a graph that displays database statistics. You could capture a bitmap of the graph in order to compare it with a bitmap of the graph from a different version of your application. If there is a difference between the graph captured for expected results and the one captured during the test run, WinRunner generates a bitmap that shows the difference, pixel by pixel.



In the expected graph, captured when the test was created, 25 tickets were sold.



In the actual graph, captured during the test run, 27 tickets were sold. The last column is taller because of the larger quantity of tickets.



The difference bitmap shows where the two graphs diverged: in the height of the last column, and in the number of tickets sold.

Creating bitmap checkpoints

When working in Context Sensitive mode, you can capture a bitmap of a window, object, or of a specified area of a screen. WinRunner inserts a checkpoint in the test script in the form of either a **win_check_bitmap** or **obj_check_bitmap** statement.

To check a bitmap, you start by choosing **Insert > Bitmap Checkpoint**. To capture a window or another GUI object, you click it with the mouse. To capture an area bitmap, you mark the area to be checked using a crosshairs mouse pointer. Note that when you record a test in Analog mode, you should press the CHECK BITMAP OF WINDOW softkey or the CHECK BITMAP OF SCREEN AREA softkey to create a bitmap checkpoint. This prevents WinRunner from recording extraneous mouse movements. If you are programming a test, you can also use the Analog function **check_window** to check a bitmap.

If the name of a window or object varies each time you run a test, you can define a regular expression in the GUI Map Editor. This instructs WinRunner to ignore all or part of the name. For more information on using regular expressions in the GUI Map Editor, see “Editing the GUI Map.”

You can include your bitmap checkpoint in a loop. If you run your bitmap checkpoint in a loop, the results for each iteration of the checkpoint are displayed in the test results as separate entries. The results of the checkpoint can be viewed in the Test Results window. For more information, see “Analyzing Test Results.”

Handling differences in display drivers

A bitmap checkpoint on identical bitmaps could fail if different display drivers are used when you create the checkpoint and when you run the test, because different display drivers may draw the same bitmap using slightly different color definitions. For example, white can be displayed as RGB (255,255,255) with one display driver and as RGB (231,231,231) with another.

You can configure WinRunner to treat such colors as equal by setting the maximum percentage color difference that WinRunner ignores.

To set the ignorable color difference level:

1. Open *wrun.ini* from the <WinRunner installation folder>\dat folder.
 2. Adding the XR_COLOR_DIFF_PRCNT= parameter to the [WrCfg] section.
 3. Enter the value indicating the maximum percentage difference to ignore.
- In the example described above the difference between each RGB component (255:231) is about 9.4%, so setting the XR_COLOR_DIFF_PRCNT parameter to 10 forces WinRunner to treat the bitmaps as equal:

[WrCfg] XR_COLOR_DIFF_PRCNT=10

Setting Bitmap Checkpoint and Capture Options

You can instruct WinRunner to send an e-mail to selected recipients each time a bitmap checkpoint fails and you can instruct WinRunner to capture a bitmap of your window or screen when any checkpoint fails. You set these options in the General Options dialog box.

You can also insert a statement in your script that instructs WinRunner to capture a bitmap of your window or screen based at a specific point in your test run.

To instruct WinRunner to send an e-mail message when a bitmap checkpoint fails:

1. Choose **Tools > General Options**. The General Options dialog box opens.
2. Click the **Notifications** category in the options pane. The notification options are displayed.
3. Select **Bitmap checkpoint failure**.
4. Click the **Notifications > E-mail** category in the options pane. The e-mail options are displayed.
5. Select the **Active E-mail service** option and set the relevant server and sender information.
6. Click the **Notifications > Recipient** category in the options pane. The e-mail recipient options are displayed.
7. Add, remove, or modify recipient details as necessary to set the recipients to whom you want to send an e-mail message when a bitmap checkpoint fails. The e-mail contains summary details about the test and the bitmap checkpoint, and gives the file names for the expected, actual, and difference images.

To instruct WinRunner to capture a bitmap when a checkpoint fails:

1. Choose **Tools > General Options**. The General Options dialog box opens.
2. Click the **Run > Settings** category in the options pane. The run settings options are displayed.
3. Select **Capture bitmap on verification failure**.

4. Select **Window**, **Desktop**, or **Desktop area** to indicate what you want to capture when checkpoints fail.
5. If you select **Desktop area**, specify the coordinates of the area of the desktop that you want to capture.

When you run your test, the captured bitmaps are saved in your test results folder.

To capture a bitmap during the test run:

Enter a **win_capture_bitmap** or **desktop_capture_bitmap** statement. Use the following syntax:

```
win_capture_bitmap(image_name [, window, x, y, width, height]);
```

or

```
desktop_capture_bitmap (image_name [, x, y, width, height]);
```

Enter only the desired image name in the statement. Do not include a folder path or extension. The bitmap is automatically stored with a **.bmp** extension in a subfolder of the test results folder.

Checking window and object bitmaps:

You can capture a bitmap of any window or object in your application by pointing to it. The method for capturing objects and for capturing windows is identical. You start by choosing **Insert > Bitmap Checkpoint > For Object/Window**. As you pass the mouse pointer over the windows of your application, objects and windows flash. To capture a window bitmap, you click the window's title bar. To capture an object within a window as a bitmap, you click the object itself.

Note that during recording, when you capture an object in a window that is not the active window, WinRunner automatically generates a **set_window** statement.

To capture a window or object as a bitmap:

- Choose **Insert > Bitmap Checkpoint > For Object/Window** or click the **Bitmap Checkpoint for Object/Window** button on the User toolbar. Alternatively, if you are recording in Analog mode, press the CHECK BITMAP OF OBJECT/WINDOW softkey.

The WinRunner window is minimized, the mouse pointer becomes a pointing hand, and a help window opens.

- Point to the object or window and click it. WinRunner captures the bitmap and generates a **win_check_bitmap** or **obj_check_bitmap** statement in the script.

The TSL statement generated for a window bitmap has the following syntax:

```
win_check_bitmap ( object, bitmap, time );
```

For an object bitmap, the syntax is:

obj_check_bitmap (*object, bitmap, time*);

For example, when you click the title bar of the main window of the Flight Reservation application, the resulting statement might be:

```
win_check_bitmap ("Flight Reservation", "Img2", 1);
```

However, if you click the Date of Flight box in the same window, the statement might be:

```
obj_check_bitmap ("Date of Flight:", "Img1", 1);
```

checking area bitmaps

You can define any rectangular area of the screen and capture it as a bitmap for comparison. The area can be any size: it can be part of a single window, or it can intersect several windows. The rectangle is identified by the coordinates of its upper left and lower right corners, relative to the upper left corner of the window in which the area is located. If the area intersects several windows or is part of a window with no title (for example, a popup window), its coordinates are relative to the entire screen (the root window).

To capture an area of the screen as a bitmap:

1. Choose **Insert > Bitmap Checkpoint > For Screen Area** or click the **Bitmap Checkpoint for Screen Area** button. Alternatively, if you are recording in Analog mode, press the CHECK BITMAP OF SCREEN AREA softkey.

The WinRunner window is minimized, the mouse pointer becomes a crosshairs pointer, and a help window opens.

2. Mark the area to be captured: press the left mouse button and drag the mouse pointer until a rectangle encloses the area; then release the mouse button.
3. Press the right mouse button to complete the operation. WinRunner captures the area and generates a **win_check_bitmap** statement in your script.

The **win_check_bitmap** statement for an area of the screen has the following syntax:

win_check_bitmap (*window, bitmap, time, x, y, width, height*);

For example, when you define an area to check in the Flight Reservation application, the resulting statement might be:

```
win_check_bitmap ("Flight Reservation", "Img3", 1, 9, 159, 104, 88);
```

6.g) How to Create Data-Driven Tests which supports to run a single test on several sets of data from a data table.

Why Data Driven Testing? Frequently we have multiple data sets which we need to run the same tests on. To create an individual test for each data set is a lengthy and time-consuming process.

Data Driven Testing framework resolves this problem by keeping the data separate from Functional tests. The same test script can execute for different combinations of input test data and generate test results.

Example:

For example, we want to test the login system with multiple input fields with 1000 different data sets.

To test this, you can take following different approaches:

Approach 1) Create 1000 scripts one for each dataset and runs each test separately one by one.

Approach 2) Manually change the value in the test script and run it several times.

Approach 3) Import the data from the excel sheet. Fetch test data from excel rows one by one and execute the script.

In the given three scenarios first two are laborious and time-consuming. Therefore, it is ideal to follow the third approach.

Thus, the third approach is nothing but a Data-Driven framework.

The procedure for converting a test to a data-driven test is composed of the following main steps:

1. Replacing fixed values in checkpoint statements and in recorded statements with parameters, and creating a data table containing values for the parameters. This is known as parameterizing the test.
2. Adding statements and functions to your test so that it will read from the data table and run in a loop while it reads each iteration of data.
3. Adding statements to your script that open and close the data table.
4. Assigning a variable name to the data table (mandatory when using the DataDriver wizard and otherwise optional).

You can use the DataDriver wizard to perform these steps, or you can modify your test script manually.

Creating a Data-Driven Test with the DataDriver Wizard

You can use the DataDriver wizard to convert your entire script or a part of your script into a data-driven test. For example, your test script may include recorded operations, checkpoints, and other statements which do not need to be repeated for multiple sets of data. You need to parameterize only the portion of your test script that you want to run in a loop with multiple sets of data.

To create a data-driven test:

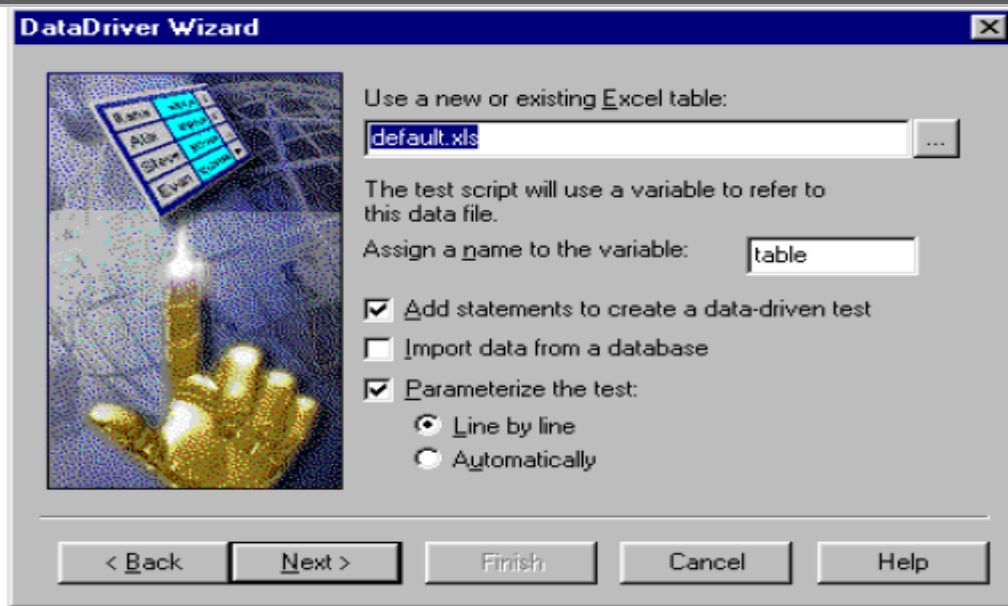
1. If you want to turn only part of your test script into a data-driven test, first select those lines in the test script.
2. Choose **Table > Data Driver Wizard**.
 - If you selected part of the test script before opening the wizard, proceed to step 3.
 - If you did not select any lines of script, the following screen opens:



If you want to turn only part of the test into a data-driven test, click **Cancel**. Select those lines in the test script and reopen the DataDriver wizard.

If you want to turn the entire test into a data-driven test, click **Next**.

3. The following wizard screen opens:



The **Use a new or existing Excel table** box displays the name of the Excel file that WinRunner creates, which stores the data for the data-driven test. Accept the default data table for this test, enter a different name for the data table, or use the browse button to locate the path of an existing data table. By default, the data table is stored in the test folder.

In the **Assign a name to the variable** box, enter a variable name with which to refer to the data table, or accept the default name, “table.”

At the beginning of a data-driven test, the Excel data table you selected is assigned as the value of the table variable. Throughout the script, only the table variable name is used. This makes it easy for you to assign a different data table to the script at a later time without making changes throughout the script.

Choose from among the following options:

- **Add statements to create a data-driven test:** Automatically adds statements to run your test in a loop: sets a variable name by which to refer to the data table; adds braces ({ and }), a **for** statement, and a **ddt_get_row_count** statement to your test script selection to run it in a loop while it reads from the data table; adds **ddt_open** and **ddt_close** statements to your test script to open and close the data table, which are necessary in order to iterate rows in the table.

Note that you can also add these statements to your test script manually. For more information and sample statements, see “Adding Statements to Your Test Script to Open and Close the Data Table and Run Your Test in a Loop”

If you do not choose this option, you will receive a warning that your data-driven test must contain a loop and statements to open and close your data table.

- **Import data from a database:** Imports data from a database. This option adds **ddt_update_from_db**, and **ddt_save** statements to your test script after the **ddt_open** statement. For more information, see “Importing Data from a Database”

You can either manually import data from a database by specifying the SQL statement, or you can import data using Microsoft Query or Data Junction. Note that in order to import data using Microsoft Query or Data Junction, either Microsoft Query or Data Junction must be installed on your machine. You can install Microsoft Query from the custom installation of Microsoft Office. Note that Data Junction is not automatically included in your WinRunner package. To purchase Data Junction, contact your Mercury Interactive representative. For detailed information on working with Data Junction, refer to the documentation in the Data Junction package.

- **Parameterize the test:** Replaces fixed values in selected checkpoints and in recorded statements with parameters, using the **ddt_val** function, and in the data table, adds columns with variable values for the parameters.

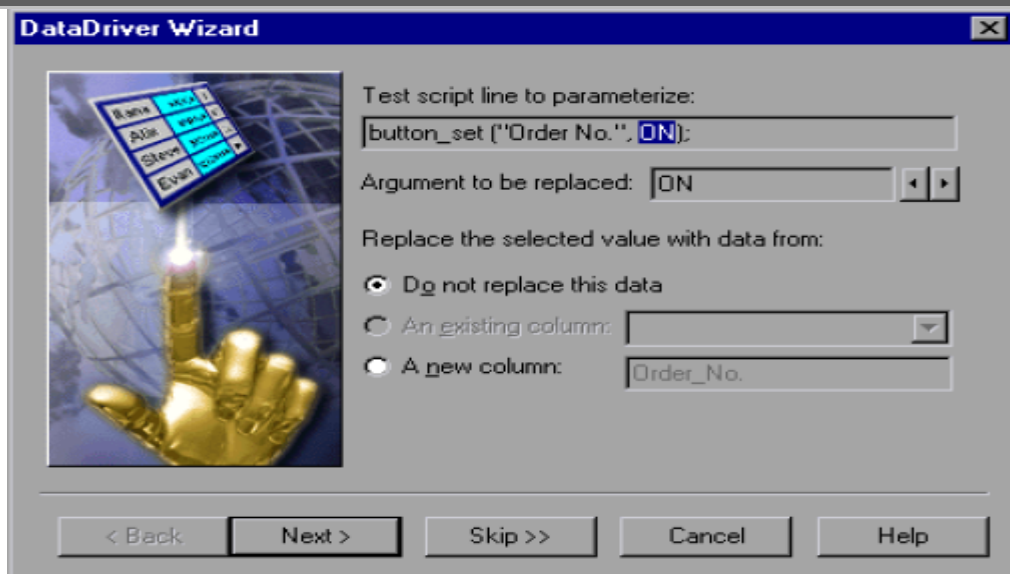
Line by line: Opens a wizard screen for each line of the selected test script, which enables you to decide whether to parameterize a particular line, and if so, whether to add a new column to the data table or use an existing column when parameterizing data.

Automatically: Replaces all data with **ddt_val** statements and adds new columns to the data table. The first argument of the function is the name of the column in the data table. The replaced data is inserted into the table.

Click **Next**.

Note that if you did not select any check boxes, only the **Cancel** button is enabled.

4. If you selected the **Import data from a database** check box in the previous screen, continue with “Importing Data from a Database”. Otherwise, the following wizard screen opens:



The **Test script line to parameterize** box displays the line of the test script to parameterize. The highlighted value can be replaced by a parameter.

The **Argument to be replaced** box displays the argument (value) that you can replace with a parameter. You can use the arrows to select a different argument to replace.

Choose whether and how to replace the selected data:

- **Do not replace this data:** Does not parameterize this data.
- **An existing column:** If parameters already exist in the data table for this test, select an existing parameter from the list.
- **A new column:** Creates a new column for this parameter in the data table for this test. Adds the selected data to this column of the data table. The default name for the new parameter is the logical name of the object in the selected TSL statement above. Accept this name or assign a new name.

In the sample Flight application test script shown earlier, there are several statements that contain fixed values entered by the user.

In this example, a new data table is used, so no parameters exist yet. In this example, for the first parameterized line in the test script, the user clicks the **Data from a new parameter** radio button. By default, the new parameter is the logical name of the object. You can modify this name. In the example, the name of the new parameter was modified to “Date of Flight.”

The following line in the test script:

```
edit_set ("Edit", "6");
```

is replaced by:

```
edit_set("Edit",ddt_val(table,"Edit"));
```

The following line in the test script:

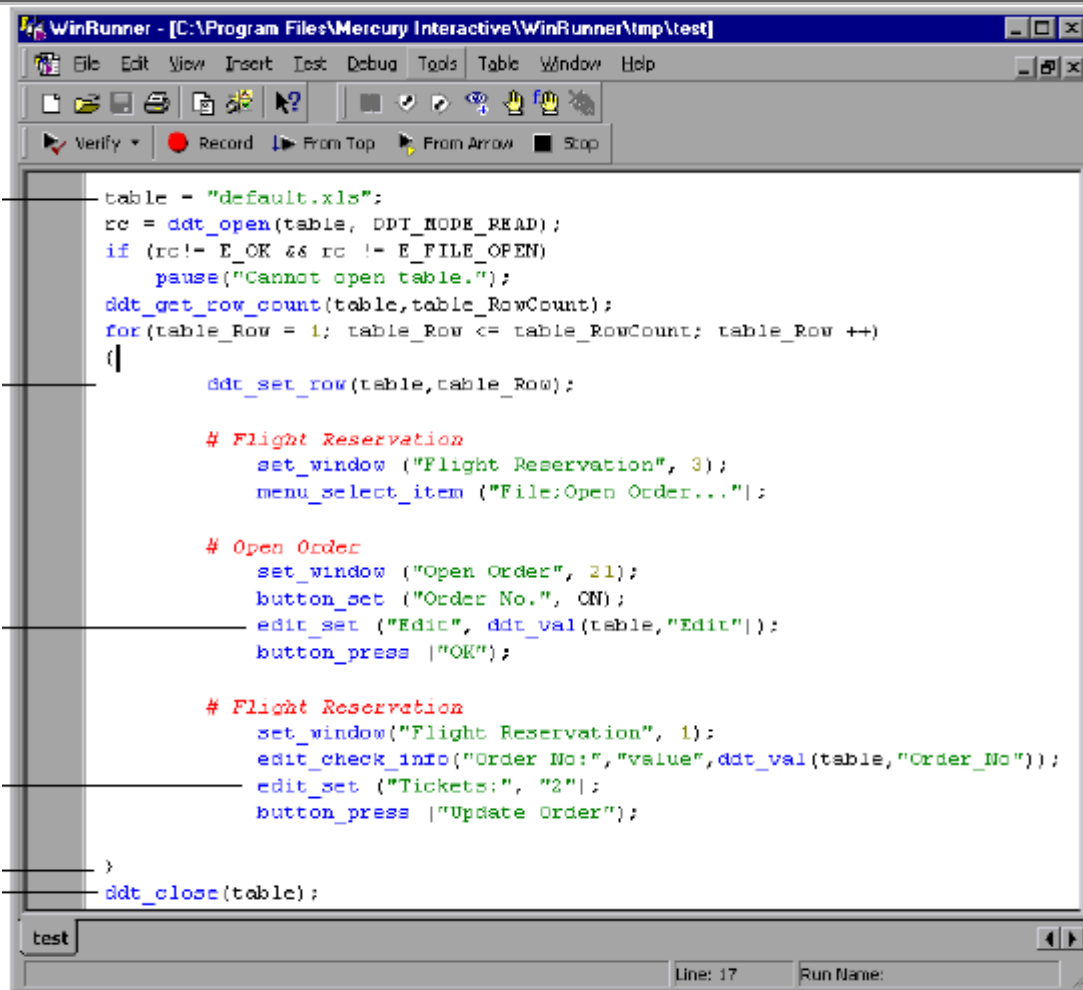
```
edit_check_info("Order No:","value",6);
```

is replaced by:

```
edit_check_info("Order No:","value",ddt_val(table,"Order_No"));
```

- To parameterize additional lines in your test script, click **Next**. The wizard displays the next line you can parameterize in the test script selection. Repeat the above step for each line in the test script selection that can be parameterized. If there are no more lines in the selection of your test script that can be parameterized, the final screen of the wizard opens.
- To proceed to the final screen of the wizard without parameterizing any additional lines in your test script selection, click **Skip**.
- 5. The final screen of the wizard opens.
 - If you want the data table to open after you close the wizard, select **Show data table now**.
 - To perform the tasks specified in previous screens and close the wizard, click **Finish**.
 - To close the wizard without making any changes to the test script, click **Cancel**.

Once you have finished running the DataDriver wizard, the sample test script for the example is modified, as shown below:



If you open the data table (**Table > Data Table**), the **Open or Create a Data Table** dialog box opens. Select the data table you specified in the DataDriver wizard. When the data table opens, you can see the entries made in the data table and edit the data in the table.

For the previous example, the following entry is made in the data table.

	Edit	Order_No	C	D	E
1	6	6			
2					
3					
4					

Creating a Data-Driven Test Manually

You can create a data-driven test manually, without using the DataDriver wizard. Note that in order to create a data-driven test manually, you must complete all the steps described below:

- defining the data table
- add statements to your test script to open and close the data table and run your test in a loop
- import data from a database (optional)
- create a data table and parameterize values in your test script

Defining the Data Table

Add the following statement to your test script immediately preceding the parameterized portion of the script. This identifies the name and the path of your data table. Note that you can work with multiple data tables in a single test, and you can use a single data table in multiple tests. For additional information, see “Guidelines for Creating a Data-Driven Test”

```
table="Default.xls";
```

Note that if your data table has a different name, substitute the correct name. By default, the data table is stored in the folder for the test. If you store your data table in a different location, you must include the path in the above statement.

For example:

```
table1 = "default.xls";
```

is a data table with the default name in the test folder.

```
table2 = "table.xls";
```

is a data table with a new name in the test folder.

```
table3 = "C:\Data-Driven Tests\Another Test\default.xls";
```

is a data table with the default name and a new path. This data table is stored in the folder of another test.

Adding Statements to Your Test Script to Open and Close the Data Table and Run Your Test in a Loop

Add the following statements to your test script immediately following the table declaration.

```
rc=ddt_open (table);  
if (rc!= E_OK && rc != E_FILE_OPEN)  
    pause("Cannot open table.");
```



```
ddt_get_row_count(table,table_RowCount);  
for(table_Row = 1; table_Row <= table_RowCount ;table_Row ++ )  
{  
    ddt_set_row(table,table_Row);
```

These statements open the data table for the test and run the statements between the curly brackets that follow for each row of data in the data table. Add the following statements to your test script immediately following the parameterized portion of the script:

```
}  
ddt_close (table);
```

These statements run the statements that appear within the curly brackets above for every row of the data table. They use the data from the next row of the data table to drive each successive iteration of the test. When the next row of the data table is empty, these statements stop running the statements within the curly brackets and close the data table.

Importing Data from a Database

You must add **ddt_update_from_db** and **ddt_save** statements to your test script after the **ddt_open** statement. You must use Microsoft Query to define a query in order to specify the data to import. For more information, see “Importing Data from a Database”

Parameterizing Values in a Test Script

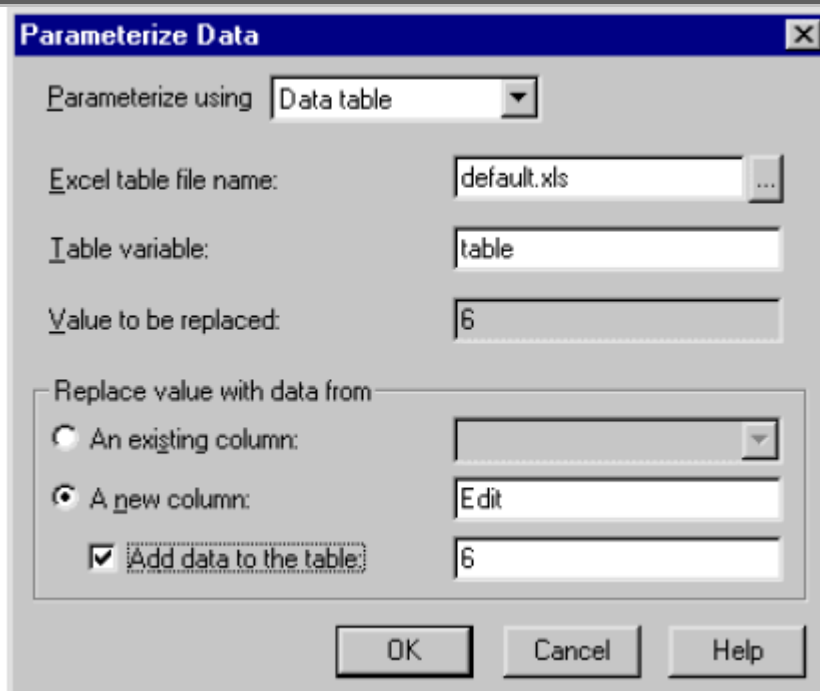
In the sample test script in “Creating a Basic Test for Conversion” there are several statements that contain fixed values entered by the user:

```
edit_set("Edit", "6");  
edit_check_info("Order No:","value",6);
```

You can use the Parameterize Data dialog box to parameterize the statements and replace the data with parameters.

To parameterize statements using a data table:

1. In your test script, select the first instance in which you have data that you want to parameterize. For example, in the first **edit_set** statement in the test script above, select: "6".
2. Choose **Table > Parameterize Data**. The Parameterize Data dialog box opens.
3. In the **Parameterize using** box, select **Data table**.



4. In the **Excel table file name** box, you can accept the default name and location of the data table, enter the different name for the data table, or use the browse button to locate the path of a data table. Note that by default the name of the data table is default.xls, and it is stored in the test folder. If you previously worked with a different data table in this test, then it appears here instead.

Click **A new column**.

WinRunner suggests a name for the parameter in the box. You can accept this name or choose a different name. WinRunner creates a column with the same name as the parameter in the data table.

The data with quotation marks that was selected in your test script appears in the **Add the data to the table** box.

- If you want to include the data currently selected in the test script in the data table, select the **Add the data to the table** check box. You can modify the data in this box.
- If you do not want to include the data currently selected in the test script in the data table, clear the **Add the data to the table** check box.
- You can also assign the data to an existing parameter, which assigns the data to a column already in the data table. If you want to use an existing parameter, click **An existing column**, and select an existing column from the list.

5. Click **OK**.

In the test script, the data selected in the test script is replaced with a **ddt_val** statement which contains the name of the table and the name of the parameter you created, with a corresponding column in the data table.

In the example, the value "6" is replaced with a **ddt_val** statement which contains the name of the table and the parameter "Edit", so that the original statement appears as follows:

```
edit_set ("Edit",ddt_val(table,"Edit"));
```

In the data table, a new column is created with the name of the parameter you assigned. In the example, a new column is created with the header Edit.

6. Repeat steps 1 to 5 for each argument you want to parameterize. For more information on the **ddt_val** function, see "Using TSL Functions with Data-Driven Tests"

6.h) How to read and check text found in GUI objects and bitmaps.

You can read the entire text contents of any GUI object or window in your application, or the text in a specified area of the screen. You can either retrieve the text to a variable, or you can compare the retrieved text with any value you specify.

To retrieve text to a variable, use the **win_get_text**, **obj_get_text**, and **get_text** functions. These functions can be generated automatically, using a **Insert > Get Text** command, or manually, by programming. In both cases, the read text is assigned to an output variable.

To read all the text in a GUI object, you choose **Insert > Get Text > From Object/Window** and click an object with the mouse pointer. To read the text in an area of an object or window, you choose **Insert > Get Text > From Screen Area** and then use a crosshairs pointer to enclose the text in a rectangle.

In most cases, WinRunner can identify the text on GUI objects automatically. However, if you try to read text and the comment “#no text was found” is inserted into the test script, this means WinRunner was unable to identify your application font. To enable WinRunner to identify text, you must teach WinRunner your application fonts and use the image text recognition mechanism.

To compare the text in a window or object with an expected text value, use the **win_check_text** or **obj_check_text** functions.

Reading All the Text in a Window or an Object

You can read all the visible text in a window or other object using **win_get_text** or **obj_get_text**.

To read all the visible text in a window or an object:

1. Choose **Insert > Get Text > From Object/Window** or click the **Get Text from Object/Window** button on the User toolbar. Alternatively, if you are recording in Analog mode, press the GET TEXT FROM OBJECT/WINDOW softkey.

WinRunner is minimized, the mouse pointer becomes a pointing hand, and a Help window opens.

2. Click the window or object. WinRunner captures the text in the object and generates a **win_get_text** or **obj_get_text** statement.

In the case of a window, this statement has the following syntax:

win_get_text (window, text);

The *window* is the name of the window. The *text* is an output variable that holds all of the text displayed in the window. To make your script easier to read, this text is inserted into the script as a comment when the function is recorded.

For example, if you choose **Insert > Get Text > From Object/Window** and click on the Windows Clock application, a statement similar to the following is recorded in your test script:

```
# Clock settings 10:40:46 AM 8/8/95  
win_get_text("Clock", text);
```

In the case of an object other than a window, the syntax is as follows:

```
obj_get_text ( object, text );
```

The parameters of **obj_get_text** are identical to those of **win_get_text**.

Reading the Text from an Area of an Object or a Window

The **win_get_text** and **obj_get_text** functions can be used to read text from a specified area of a window or other GUI object.

To read the text from an area of a window or an object:

1. Choose **Insert > Get Text > From Screen Area** or click the **Get Text from Screen Area** button on the User toolbar. Alternatively, if you are recording in Analog mode, press the GET TEXT FROM SCREEN AREA softkey.

WinRunner is minimized and the recording of mouse and keyboard input stops. The mouse pointer becomes a crosshairs pointer.

2. Use the crosshairs pointer to enclose the text to be read within a rectangle. Move the mouse pointer to one corner of the text you want to capture. Press and hold down the left mouse button. Drag the mouse until the rectangle encompasses the entire text, then release the mouse button. Press the right mouse button to capture the string.

You can preview the string before you capture it. Press the right mouse button before you release the left mouse button. (If your mouse has three buttons, release the left mouse button after drawing the rectangle and then press the middle mouse button.) The string appears under the rectangle or in the upper left corner of the screen.

WinRunner generates a **win_get_text** statement with the following syntax in the test script:

```
win_get_text ( window, text, x1,y1,x2,y2 );
```

For example, if you choose Get Text > Area and use the crosshairs to enclose only the date in the Windows Clock application, a statement similar to the following is recorded in your test script:

```
win_get_text ("Clock", text, 38, 137, 166, 185); # 8/13/95
```

The *window* is the name of the window. The *text* is an output variable that holds all of the captured text. *x1,y1,x2,y2* define the location from which to read text, relative to the specified window. When the function is recorded, the captured text is also inserted into the script as a comment.

The comment occupies the same number of lines in the test script as the text being read occupies on the screen. For example, if three lines of text are read, the comment will also be three lines long. You can also read text from the screen by programming the Analog TSL function **get_text** into your test script.

6.i) How to create a batch test that automatically runs the tests.

A batch test is a test script that calls other tests. You program a batch test by typing call statements directly into the test window and selecting the **Run in batch mode** option in the **Run** category of the General Options dialog box before you execute the test.

A batch test may include programming elements such as loops and decisionmaking statements. Loops enable a batch test to run called tests a specified number of times. Decision-making statements such as if/else and switch condition test execution on the results of a test called previously by the same batch script. See “Enhancing Your Test Scripts with Programming,” for more information.

For example, the following batch test executes three tests in succession, then loops back and calls the tests again. The loop specifies that the batch test should call the tests ten times.

```
for(i=0;i<10;i++)  
{  
    call"c:\\pbtests\\open";  
    call"c:\\pbtests\\setup";  
    call"c:\\pbtests\\save";  
}
```

To enable a batch test:

1. Choose **Tools > General Options**.
The General Options dialog box opens.
2. Click the **Run** category.
3. Select the **Run in batch mode** check box.

6. J. How to update the GUI object descriptions which in turn supports test scripts as the application changes?

WinRunner uses the GUI map to identify and locate GUI objects in your application. If the GUI of your application changes, you must update object descriptions in the GUI map so you can continue to use existing tests. You can update the GUI map in two ways:

- during a test run, using the Run wizard
- at any time during the testing process, using the GUI Map Editor

The Run wizard opens automatically during a test run if WinRunner cannot locate an object in the application being tested. It guides you through the process of identifying the object and updating its description in the GUI map. This ensures that WinRunner will find the object in subsequent test runs.

You can also:

- manually edit the GUI map using the GUI Map Editor
- modify the logical names and physical descriptions of objects, add new descriptions, and remove obsolete descriptions
- move or copy descriptions from one GUI map file to another

Before you can update the GUI map, the appropriate GUI map files must be loaded.

You can load files by using the *GUI_load* statement in a test script or by choosing *File Open* in the GUI Map Editor. See “Working in the Global GUI Map File Mode,” for more information.