

# eggPlant

v11.0 WINDOWS

## Using EggPlant



Copyright 2011 TestPlant Inc.  
Using Eggplant

### **Trademarks**

Eggplant, the Eggplant logos, TestPlant, and the TestPlant logo are trademarks or registered trademarks of TestPlant Inc.

*Eggplant Reference Manual, Eggplant: Getting Started, Using Eggplant, SenseTalk Reference Manual, and Eggplant: RiTA Manual* are copyrights of TestPlant Inc.

SenseTalk is a trademark or registered trademark of Thoughtful Software, Inc.

Apple, Mac, Macintosh, Mac OS X, and QuickTime are trademarks or registered trademarks of Apple Computer, Inc.

Windows, and Window XP are trademarks or registered trademarks of Microsoft Corporation.

# Contents

<b>Preface</b> .....	<b>6</b>
About This Manual .....	6
Help and Additional Information .....	6
<b>Capturing Images</b> .....	<b>7</b>
Setting up Image Captures.....	7
Identifying Good Images .....	7
Using the Hot Spot.....	8
The Relative Hot Spot.....	8
Moving the Hot Spot.....	8
Choosing the Best Search Type.....	9
Text search types .....	9
Capturing Tooltips and Other Transient GUI Elements .....	10
Keeping the tooltip during script execution .....	10
Comparing Images with Graphics Software .....	10
Reusing Captured Images.....	11
Insert Pop-up Menu.....	11
Drag-and-drop.....	11
Copy and paste.....	11
<b>Typing on the SUT</b> .....	<b>12</b>
Typing in Live Mode .....	12
Automating Keystrokes .....	12
Examples: The TypeText Command .....	12
Recording TypeText statements.....	12
<b>Finding Text</b> .....	<b>14</b>
Using text property lists .....	14
Inserting a text property list into your script: Step-by-Step.....	15
What is included in a text property list?.....	15
Using the Generic (OCR) text platform .....	17
When to use a generic text property list.....	17
How the OCR text engine works .....	17
Improving the speed of text searches .....	17
Using platform-specific (TIG) text property lists .....	18
When to use a platform-specific (TIG) text property list.....	18
Text Images in a script .....	18
Reusable Text Styles.....	19
More about platform-specific text platforms .....	19
Text Image generators (TIGs) .....	20
TIG Scripts .....	20
Text Image Mark-ups.....	23
<b>Finding Images</b> .....	<b>24</b>
Finding an Image with several possible states.....	24
Avoiding timing problems .....	24
The WaitFor Command.....	24
Remote Work Interval.....	25

Streamlining with Image Functions .....	27
FoundImageLocation() .....	27
ImageFound().....	27
ImageLocation() .....	27
Searching Part of the Screen .....	28
Optimizing Script Performance .....	29
<b>Re-using Code .....</b>	<b>30</b>
Scripts .....	30
Handlers .....	30
Returning Results of a Handler .....	31
Handlers as Commands or Functions .....	31
Calling a handler from another script .....	31
Testing Cross-Platform Applications .....	32
Creating platform and language-specific suites .....	32
Using multilingual text Images .....	32
Running from a Master Script.....	34
<b>Gathering and Using Data .....</b>	<b>36</b>
Reading text on the SUT .....	36
Reading text within a rectangle .....	36
Reading text near a point .....	36
Using the SUT clipboard .....	36
Data-driven Testing .....	37
Creating a Data File .....	38
Reading and Validating a Data File .....	38
Timing Script Events.....	40
<b>Reading Results .....</b>	<b>41</b>
LogFile.txt.....	41
RunHistory.csv.....	41
SuiteStatistics.csv .....	42
Reporting Results.....	42
Further Reading .....	42
<b>Organizing Your Testing .....</b>	<b>43</b>
Have a Plan.....	43
Think in Terms of End-User Functionality.....	43
Be Aware of the Visible Interface.....	43
Establish a Consistent Interface.....	44
Record the System State .....	44
Organizing your Images .....	45
Folder structure .....	45
Naming conventions.....	45
Sample naming conventions .....	46
<b>Connecting under Special Circumstances .....</b>	<b>47</b>
Direct Connections .....	47
Setting up a Direct Connection on Each Machine: Step-by-Step.....	47
Reverse VNC Connections .....	47
Preparing EggPlant for Reverse Connections: Step-by-Step .....	48

**Troubleshooting Connection Issues .....49**

# Preface

## About This Manual

---

Many of the articles in Using EggPlant provide instructions for accomplishing specific tasks in EggPlant; others discuss more general issues, or bring together information that is a little bit outside the realm of a reference manual. You can read this manual from beginning to end, or just refer to individual sections when a need arises.

## Help and Additional Information

---

The following manuals are available through the EggPlant Help menu and the downloads page of the [TestPlant web site](#).

*The EggPlant: Reference Manual* describes the EggPlant interface and scripting processes, and the SenseTalk commands, functions, and global properties that are unique to EggPlant.

*Using EggPlant* is a collection of articles that cover a wide range of EggPlant topics.

*EggPlant Tutorials* is a series of five tutorials that introduce the scripting environment and often-used commands and functions.

*The SenseTalk Reference Manual* is a comprehensive guide to the SenseTalk scripting language used in EggPlant.

For EggPlant updates, news, discussion forums, and all available support resources, please visit [TestPlant support](#).

### Capturing Images

Images are essential to EggPlant scripting.

The articles in this section can help you get started with basic techniques for capturing images.

# Capturing Images

## Setting up Image Captures

---

In the Viewer window, you can move the capture area by clicking the desired location, dragging the capture area, or pressing the arrow keys. (Pressing an arrow key alone moves the capture area one pixel in the given direction; to move the capture area in 10-pixel increments, hold down Shift as you press an arrow key.)

**To resize the capture area**, drag the edges or corners of the capture area, or press Alt + arrow keys. (Add the shift key to resize the capture area in 10-pixel increments.)

**To move the image hot spot**, hold down Control and click the desired location, drag the hot spot, or press the arrow keys. (Add the shift key to move the hot spot in 10-pixel increments.)

## Identifying Good Images

---

An easy way to keep EggPlant scripts as robust as possible is to capture images with just enough content to uniquely identify an interface element. For example, here is an image of a folder on a desktop:



*Folder with desktop background*

If you capture some of the desktop in your image, you always have to rely on the desktop color to be the same for a match. Even if you can always count on the desktop being the same color, in this example, it is shaded when the folder icon is selected:



*Selected folder with dark blue area in the background*

Of course, you could always capture an image of the folder with the selected background as well, but since the icon itself does not change, this is an unnecessary complication. If you capture the icon without the desktop in the background, you can cover both states with one image:



*Folder with no background*

This image may look less like a folder icon to you, but shape isn't especially important to EggPlant. As long as the inside of the image does not match anything else on the screen, the edges are not important.

When you have the Capture Area in roughly the size and position you need, you can use the Arrow keys to fine tune it:

- To nudge the Capture Area one pixel at a time, press the Arrow keys.
- To adjust the size of the Capture Area one pixel at a time, press Alt-Arrow keys.
- To nudge the Hot Spot one pixel at a time, use Control-Arrow keys.

Add Shift to any of these shortcuts to make the adjustments in ten-pixel increments.

## Using the Hot Spot

---

The Hot Spot is the point that EggPlant targets for mouse actions such as Click, DoubleClick, and MoveTo. It is also the point used to describe the location of an image found in the Viewer window.

The position of the Hot Spot is defined as an (x, y) offset relative to the upper-left corner of the captured image. For example, if you capture an image that is 20 pixels wide and 10 pixels high, and you leave the Hot Spot at its default location in the center of the image, the Hot Spot location is reported as (10, 5)—10 pixels to the right of and 5 pixels down from the upper-left corner.

## The Relative Hot Spot

Although the Hot Spot is associated with an image, it doesn't have to be *inside* the image. For example, if you need to select text in a text field, there is no way to do an image match unless you know in advance what the text is going to be. Instead, you can capture an image of only the text field's label, and set the Hot Spot a few pixels over, where the actual text field begins.

Another example is a Country pop-up menu that can display any one of dozens of countries. Instead of capturing an image each possibility, you could capture the label *Country* with the Hot Spot on the actual pop-up menu. Since the pop-up menu is not included in the image, the country that is displayed has no bearing on the image match.

## Moving the Hot Spot

You can change an image's Hot Spot in the Viewer window, the Capture Image panel, an image property list in a script, and in the Images pane of the Suite Editor.

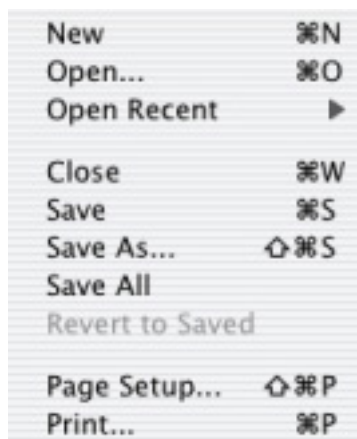
There are several easy ways to move a Hot Spot:

- Control-click in or around an image.
- Control-drag the red cross hairs that indicate the Hot Spot.
- Press Control-Arrow keys to nudge the Hot Spot one pixel at a time.
- Press Shift-Control-Arrow to nudge the Hot Spot ten pixels at a time.
- In the Viewer window, resize the Capture Area to snap the Hot Spot back to the center of the image.

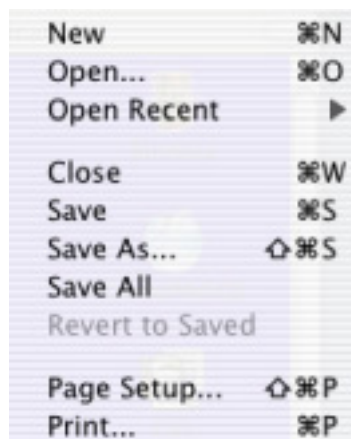
## Choosing the Best Search Type

When you save an image, the Capture Image panel gives you a choice of five search types. EggPlant tries to automatically detect the appropriate search type for your image, but in some cases you may want to override the default setting.

Typically, EggPlant chooses the Tolerant setting whenever it can. The Tolerant setting works well for most images, and allows for some variation in the colors of the image when it is found. A good example is a semi-transparent window or menu. In the images below, you can see objects in the background through the menu on the right. The Tolerant setting allows EggPlant to accurately locate menu items, even if the screen behind the menu is in a different state than it was when the image was captured.

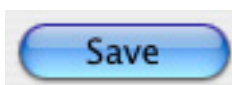


*Basic menu appearance*



*Menu with desktop items showing through*

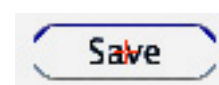
Pulsing buttons, such as the Save buttons in Mac OS X, can also be detected automatically by EggPlant. The Pulsing setting causes EggPlant to use a “mask” on the image to filter out the pixels that are changing.



*Light end of pulse*



*Dark end of pulse*



*Captured image with Pulsing mask applied*

## Text search types

The last two search types are Text and Text&Pulsing. These search types are designed to deal with text and buttons that sometimes change their appearance subtly; primarily due to Mac OS X text antialiasing. If you are testing against Mac OS X systems, using the Text (or Text&Pulsing) search type can make a big difference in your images that include text.

**Note:** The Text and Text&Pulsing search types are never selected automatically by EggPlant.

If an EggPlant script suddenly fails to find an image that it found on a previous run, you can always try changing the images search type in the Info panel of the Images pane. Search type changes are saved immediately, and the new search type is used the next time a script calls that image.

## Capturing Tooltips and Other Transient GUI Elements

---

The following steps make it much easier to capture images of tooltips (and other objects that come and go quickly).

- 1 Move the mouse to a position that brings up the tooltip, and press Control to toggle into Capture Mode.
- 2 Close your connection to the SUT, to “freeze” the Viewer window before the tooltip disappears.
- 3 In the Connection List, right-click the name of the SUT, and choose View Window.
- 4 Capture your image by clicking the Capture Image button or double-clicking the Capture Area. (The command buttons are not available when there is no connection open.)

### Keeping the tooltip during script execution

By default, if EggPlant fails find an image it is looking for on the first search, it moves the mouse to the lower right-hand corner of the screen, to make sure the image is not being obscured by the cursor.

When you are trying to capture a tooltip, moving the mouse out of the way actually *causes* a problem, because the tooltip disappears as soon as the mouse moves away. Fortunately, you can temporarily disable this behavior by setting the `shouldRepositionMouse` global property to *no*.

### Examples: Changing the value of a global property

```
setOption shouldRepositionMouse, //SetOption command, followed by the global
    No                          property name, and the new value

set the shouldRepositionMouse to //Set command, followed by "the" and the
    No                          global property name, then "to" and the
                               new value
```

To write a handler that changes a global property as needed and then changes it back to its starting value, see the [FastImageFound](#) example in [Optimizing Script Performance](#).

## Comparing Images with Graphics Software

---

If EggPlant continuously fails to find a particular image, and you have ruled out search type and timing problems, it may be that the SUT is not displaying the image consistently.

In these rare cases, it is sometimes helpful (or at least informative) to be able to perform a detailed comparison of the captured image to one being displayed by the SUT.

The procedure outlined below describes how to view the differences between these images on a pixel-by-pixel basis. (PhotoShop Elements is used here, but you can convert this technique to whichever graphics program you typically use.)

- 1 **Open both images.** In the graphics program, open the saved image and the `Screen_Error` file for a script run that failed because of this image.

- 2 **Overlap the images.** Copy the saved image, and paste it on top of the Screen\_Error file. (In PhotoShop, the copied image is automatically pasted into a separate layer.)
- 3 **Align the images.** Use the Move tool to drag the saved image to its corresponding location in the Screen\_Error file. Then zoom in and adjust the image further by nudging it with the arrow keys.
- 4 **Highlight the differences.** In the Layers palette, click the Blending Mode pop-up menu and choose *Difference*.
- 5 **Evaluate the differences.** Pixels that match perfectly are displayed as black. For other pixels, look at the Info panel to see the RGB values for each image. The greatest difference between the three values is the tolerance EggPlant must allow to consider that pixel a match in both images. (For example, if a pixel in the saved image has RGB values (99, 99, 135), and the same pixel in the Screen\_Error has (100, 100, 150), the search tolerance must be at least 15- the difference between the blue values.)

## Reusing Captured Images

---

When you are scripting, you often need to use the same image more than once within a suite or script. For example, you may click the File menu repeatedly, or open a number of dialogs with OK buttons.

Here are four easy ways to insert a previously-saved image into your script:

### Insert Pop-up Menu

In the Script Editor, choose a command (or *Additional Image* for no command) from the Insert pop-up menu, then select your image in the file browser. (A command you choose here is *not* performed on the SUT.)

### Drag-and-drop

Drag an image from the Images pane of the Suite Editor to the script.

### Copy and paste

Copy and paste from another part of the script.

Of course, you can always type an image name in quotation marks, too.

## Typing on the SUT

### Typing in Live Mode

When you are controlling the SUT through EggPlant, most of your keystrokes are automatically sent to the SUT. However, there are a few keystrokes and key combinations that are intercepted by the EggPlant computer before they can be sent to the SUT. (For example, on Mac OS X, *Command-Tab* changes focus to the next application.)

To send one of these keyboard shortcuts to the SUT, choose the shortcut in the *Control* menu.

### Automating Keystrokes

You can automate keystrokes on the SUT by using the `TypeText` command in your script. `TypeText` takes any number of the following parameters, separated by commas:

- **String:** Text in quotation marks, typed literally.
- **TypeText Keyword:** The name of a non-character key (such as `Escape`), or key that cannot be identified by a character alone (such as keypad numbers). See the *EggPlant Reference Manual* for a list of all `TypeText` keywords.

### Examples: The TypeText Command

```
TypeText "Sarah Smith" //Types Sarah Smith on the SUT.
TypeText "Sarah",space,"Smith" //Types Sarah Smith on the SUT.
TypeText ControlKey, AltKey, DeleteKey //Holds down Control and Alt, and
presses Delete. (Control
and Alt are released at the
end of the command.)
TypeText ShiftKey,"Sarah ", "Smith" //Holds the shift key and types
SARAH SMITH. (Shift is
released at the end of the
command.)
```

### Recording TypeText statements

In the Viewer window, you can create `TypeText` statements by recording your keystrokes. To start a `TypeText` command in Capture mode, press the spacebar or type any characters. The `TypeText` panel opens. (In Live mode or Capture mode, you can also click the `TypeText` button in the Viewer Window toolbar.)

When the `TypeText` panel is open, your keystrokes are inserted into the text field. Non-character keys (and quotation marks) are inserted as `TypeText` keywords; character keys are inserted as literal text. As you are typing, you can select and edit the text in the text field.

## Recording intercepted keystrokes

When you are using the TypeText panel, some key commands cannot be typed directly because they are intercepted by the computer. (For example, the Return key acts as an "OK" button.) To insert one of these key commands into your TypeText statement, choose the command from the "Enter Keystroke" pop-up menu.

## Finding Text

In many cases, you can find text on the SUT the same way you would find any other item on the SUT— by capturing an image and searching for it in your script. Captured images are easy to use, and very reliable.

When you can't capture an image of your text (usually because the text content can't be predicted in advance, or the possible values are too numerous to make image capture practical) you can substitute a text property list for a captured image.

Text property lists describe the content and appearance of text on the SUT. Like found images or screen coordinates, text property lists represent a particular location on the SUT screen— the place where the described text is found.

In the examples shown below, an image name and two text property lists are used to click the same image: the Help menu.

### Example: Using a captured image and text property lists in a script

```
Click "HelpMenu" //Finds the image of the Help menu; then clicks it.
```

```
Click (text:"Help", textPlatform:"Generic") //Finds the word "Help" with any appearance; then clicks it.
```

```
Click (text:"Help", textPlatform:"Win7", textStyle:"Menu") //Generates an image of the word Help in the Menu text style, finds the generated image; then clicks it.
```

## Using text property lists

You do not need to use a text property list every time you search for text in your script; in fact, the best practice is to use captured images by default.

The benefit of text property lists is that you can use them to search for dynamic text that would be difficult to capture in advance. For example:

- Text that can't be known in advance, such as real-time data on a live system.
- Text that has many possible values, such as a random item chosen from a long list.
- Text that is dynamically sized, where your captured image might not match the current rendering.

## Inserting a text property list into your script: Step-by-Step

These steps tell you how to create a text property list through the graphical user interface in the Viewer window. (You can always type or paste text property lists into your script, as well.)

- 1 In the Viewer-window toolbar, click the Find Text button. The Text Property List panel opens.
- 2 In the Text String field, type the that text string you want to find on the screen.
- 3 Choose a text platform. If you have already been using a specific platform for your current SUT, choose that platform; otherwise, choose *Generic*.
- 4 Choose a text style. Choose a text style that reflects the appearance of the text you want to find on the SUT; or set the text options independently as needed.

**Note:** If you are using a SUT-specific text platform, the text style must completely describe the text you are searching for—the font, size, text color, background color, italics, and font weight. If you are using a generic text platform, the language of the text style is the only essential property.

- 5 In the Command pop-up menu, choose a command to insert into your script with the text property list, just as you would for a captured image.

## What is included in a text property list?

A text property list always includes a text property—the text string you are searching for on the SUT. Beyond that, you can also include a text platform, text style, and overrides to the text style, described below. If you don't provide these properties, they are assumed to have the values you set in your Text preferences (or values that you set elsewhere in your script.)

### Example: Text Property Lists

```
Click (text: "Jean Jones") //Clicks the string "Jean Jones", using the
    default text platform and style
```

```
Click (text: "Jean Jones", textPlatform: "Generic") //Clicks the string
    "Jean Jones", using the generic text platform and the default text
    style
```

```
Click (text: "Jean Jones", textStyle: "member name") //Clicks the string
    "Jean Jones", using the default text platform and "member name" text
    style
```

## The Text property

The simplest text property list contains only a text property, the text string you are searching for on the SUT. The text value can be a string literal or a variable, as shown in the example below.

## The TextPlatform property

The TextPlatform property represents the text rendering of a particular SUT, usually differentiated by the SUT's operating system. (The generic text platform is an exception; it can be used with any SUT.)

When you are choosing a text platform, the best practice is to consider the options in the following order:

- 1 **Continue using a platform that works for you.** If you have already established a text platform that works for your current situation, there is no reason to stop using that text platform.
- 2 **Try the Generic text platform.** The Generic text platform is usually the easiest text platform to use, because it doesn't require any external set-up, and it doesn't require information about the text styles used on your SUT. The Generic text platform uses the OCR text engine; which reads all of the text on the screen, and parses the text to find the string you are looking for.
- 3 **Try a text platform that is specific to your SUT.** Most of the SUT-specific text platforms use text-image generators (TIGs) to render an image of your text ; then EggPlant searches for that image as if it were a regular captured image. TIG images can be as reliable as captured images, but they require that you know specific information about the text you are looking for: font name, size, text color, and background color. TestPlant provides TIGs for several widely-used operating systems. If there is no available TIG for your SUT, you can create a scripted TIG or try using an OCR text property list. (See [TIG Scripts](#).)

## The TextStyle property

Within a text platform, each text style represents a particular set of text attributes. The name of a text style usually reflects the usage of the text (e.g. title-bar text, menu text, clock text).

For the Generic text platform, choose a text style that includes the correct language for your text. (Characters that are not part of the selected language might be passed over, or interpreted incorrectly. For example, *î* might be read as *i*.) The Contrast and Case Sensitive properties are not strictly required.

For SUT-specific text platforms, text styles comprise formatting properties (font, size, text color, etc.) Your text style *must* match the text you are searching for on the SUT.

## Style-override properties

When you choose a text style in the Find Text panel, the properties of that style are shown below the names of the text platform and text style. You can change any of these properties to override the selected text style. (The change only applies to your current text property list; the defined text style does not change.)

**Example:** `Click (text: orderNumber, textStyle: "Order Form", textColor: "red")`

## Additional text properties

In addition to the text properties that are shown on the Find Text panel, you can also type the following properties into a text property list in your script:

**DPI.** (Integer.) The DPI property refers to the DPI (dots per inch) of the SUT display. The default DPI is 72; if you are having problems finding your text on the SUT, check the SUT's DPI setting, and adjust the DPI property of your text property list accordingly.

**SearchRectangle.** (Rectangle.) Like the searchRectangle image property, the searchRectangle limits your text search to the given rectangle. By default, searches are limited to the area defined by the searchRectangle global property.

**ValidCharacters.** (Text string.) The validCharacters property limits the characters that may be found by the OCR text engine. For example, given the property *validCharacters: "ABCDE"*, the OCR text engine will attempt to read only those five characters in your searchRectangle. The validCharacters property should only be used when you expect *only* a specific set of characters in your searchRectangle. If characters outside the validCharacters set appear within your searchRectangle, those characters might be read incorrectly.

**Example:** `Click (text: orderNumber, searchRectangle: (topLeft, topLeft + (100,20)), validCharacters: "1234567890")`

## Using the Generic (OCR) text platform

---

For the sake of simplicity, these articles refer to the Generic text platform. The same information can also be applied to any text platform that uses the OCR search engine.

### When to use a generic text property list

You can use a generic text property list almost any time you need to search for text on the SUT; and they are especially well-suited to the following situations:

- You have not already defined a specific text platform for your text.
- There is no ready-made text-image generator for your SUT.
- Your application-under-test uses a custom text-rendering engine.
- You do not know the font names, sizes, or colors of the text you need to find.
- The text you need to find uses dynamic letter spacing or non-standard sizing.

### How the OCR text engine works

When you search for text through a generic text property list, the OCR engine finds each instance of text on the SUT, and reads the text values to find the string you are looking for. You don't have to describe the appearance of your text, because text formatting is not taken into account.

### Improving the speed of text searches

The flexibility of OCR text searches comes with a trade-off: OCR text searches are not as fast as image searches. To keep your scripts running as efficiently as possible, follow these best practices:

- 1 **Use the search rectangle.** If you can narrow down the general location of the text you are looking for (e.g. in the taskbar, or within a particular window), limit your search to that location. (For more information, see ["Additional text properties"](#), above; or ["Searching part of the screen"](#).)
- 2 **Minimize "clutter" on the SUT.** It never hurts to keep a neat SUT (i.e. close unnecessary windows, use a simple desktop background), and when your script includes OCR text searches, a neat desktop can save you valuable seconds or even minutes of execution time.

## Using platform-specific (TIG) text property lists

Platforms that use SUT-specific text-image generators (TIGs) provide a very targeted way to search for text; they use the information in your text property list to create an image of your text as rendered by your SUT; then EggPlant searches for that image as if it were a regular captured image. TIG images can be as reliable as captured images, but they require that you know specific information about the text you are looking for: font name, size, text color, and background color. TestPlant provides TIGs for several widely-used operating systems. If there is no available TIG for your SUT, you can create a scripted TIG or try using an OCR text property list. (See [TIG Scripts](#).)

## When to use a platform-specific (TIG) text property list

As a general rule, it is easier to use the Generic text platform than to set up a platform-specific text platform; however, there are several occasions in which the latter might be preferred:

- You have already defined a specific text platform for your text; and it is working well for you.
- The formatting of your text is part of what you are testing.
- The Generic text platform does not read your current text consistently.
- You need faster text searches than you can perform with the Generic text platform.

## Text Images in a script

When you insert a Text Image into a script, you can see that it is actually a property list that contains the text and formatting information you provided.

### Example: Text-Image property list

```
Click (Text:"Help", TextSize:"12", //Searches for the word "Help" in Chi-
      TextFont:"Chicago")         cago font, 12 pt., and clicks it.
```

## Changing the content of a Text Image

When you insert a Text Image into your script, you can always edit its content (or any text attributes) in the Script Editor. In fact, when you create similar Text Images later, it is often faster to copy, paste, and edit the original Text Image than it would be to start again from the Text Image panel.

In the example shown below, a Help menu Text Image is edited to represent other menus in the same application. The new Text property can be another string (in quotation marks), or a variable (without quotation marks).

### Example: Changing the content of a Text Image

```
Click (Text:"Help", TextSize:"12", //Searches for the string "Help" in
      TextFont:"Arial")           Chicago font, 12 pt., and clicks it.

Click (Text: MenuName, TextSize:"12", //Searches for the word indicated by
      TextFont:"Arial")           the MenuName variable, and clicks
                                  it.
```

## Reusable Text Styles

The first time you are creating a particular kind of Text Image, you have to be able to identify the font, size, and color of the text you want to match. If you save this information as a *text style*, you can save yourself the trouble of having to find it again next time.

### Example: Condensing text properties into a text style

```
(Text:"Save", TextFont: "Helvetica", TextSize: // Describes the properties
    "10", TextColor: "blue", Bold: "yes")      of a button's text
(Text:"Save", TextStyle: "Button")
```

Text styles also make your scripts easier to maintain. If the appearance of your text changes, you can update all of the affected Text Images at once by editing the text style; otherwise, you would have to go through your scripts and edit each Text Image individually.

### Creating a text style: Step-by-Step

- 1 In Text preferences > Platform pop-up menu, choose the text platform for which you are creating a text style.
- 2 In the bottom section of the Text preferences pane, click the Add button, and name your text style.
- 3 Use the menus and checkboxes to select the properties of your text style.

**Tip:** If you are creating a new text style that is based on a style you already have, view the base style before you click the Add button. The text properties have the same initial values as the last style you viewed.

## More about platform-specific text platforms

Text platforms are EggPlant's way of compensating for text differences between various SUTs. Each text platform has a designated *text-image generator* (TIG) that determines which operating system renders its text. It can also have defined *text styles*, which describe the fonts used in the SUT's interface.

## Switching between text platforms: Advanced

If you are using multiple text platforms within a script, it is important to know the order in which a Text Image "looks for" the appropriate text platform.

- 1 If the Text Image has an assigned `TextPlatform` as part of its property list, that text platform is always used.

**Example:** (`Text: "Save", TextPlatform:"WindowsXP"`)

- 2 If a Text Image does not have its own `TextPlatform` property, it uses the `CurrentTextPlatform` global property. You can set or change the `CurrentTextPlatform` at any point in a script.

**Example:** (`Set the CurrentTextPlatform to "Vista"`)

- 3 If there is no `CurrentTextPlatform` global property, the Text Image uses the text platform named as the `Default` in Text preferences.

## Text Image generators (TIGs)

Since EggPlant is a cross-platform application, the system you are automating might render text differently than the computer that is running EggPlant. When you choose a TIG for your text platform, you determine which computer draws your Text Images.

There are two TIG options:

- **External.** This choice defaults to the TIG that is found on the current SUT. (To designate a TIG on a computer other than the current SUT, enter its connection information in the Host and Port fields.)
- **Scripted.** For operating systems that do not have a TIG available, you can write a script that serves as a custom TIG, and call that script as a command in other scripts.

## TIG Scripts

A "scripted text-image generator" sounds deceptively intimidating; it's really just a script that types in a text editor, then captures the text as a text image. You can use any text editor for your TIG, as long as it can produce the text that you need to match (with the same text attributes and rendering.)

## Switch connections if possible!

The purpose of a TIG is to produce text images that *looks like* the text on your SUT; the best practice is actually to generate the text on a separate system of the same type, if possible.

Whenever your main script calls a TIG, EggPlant performs actions that are outside the workflow of the main script, such as launching the text editor and moving the mouse to select text attributes. If these actions take place on the SUT, they can change the state of the SUT in ways that skew your test results. (If you are aware of these changes,

you can often compensate for them, but it's easier to avoid the problem altogether.)

When the TIG switches to a separate system to generate text images, the SUT isn't affected in any way. After the text image is generated, the TIG just has to make the SUT the active connection again, and the main script can proceed normally.

## The TIG use case

- 1 Switch the active connection to your TIG host.
- 2 On the TIG host, launch the text editor and open a document.
- 3 Set the text editor's formatting options to match the text attributes of your text image.
- 4 Type the text of your text image.
- 5 Capture the text image. (Make sure that your searchRectangle does not include any content other than text and background. Any amount of empty background space is fine.)
- 6 Restore the TIG host to the state in which you expect to find it next time.
- 7 Switch the active connection back to your SUT.

## Example: TIG for Mac OS X

```
(* This script types in TextEdit, and creates a text image from the typed
text. *)

(* Connect to the TIG Host *)
Set MainConnection to ConnectionInfo() //Saves the current connection so it
can be restored later
Connect (Name:"Mac_TIG_Host")

(* Set up TextEdit *)
Click "TextEdit" //Launches TextEdit. (New document opens on launch.)

If not imageFound("Formatting") //Looks for an icon on the font panel.
TypeText Command, "F" //Types the font panel shortcut
end if

(*Create the text of the image *)
set newTextImage to param(1) //Sets this variable to the text image property
list that's passed in from the main script
Set_Font (newTextImage) //Sets text attributes
TypeText (newTextImage.text) //Types the text value of the text image
property list

(*Set the text image's rectangle value *)
set TopLeft to imagelocation("closeWindow")+ (-12, 24)
set BottomRight to TopLeft + (200,200)
set newTextImage.rectangle to (TopLeft, BottomRight)

CaptureTextImage (newTextImage)
Connect (MainConnection) //Restores the SUT as the active connection

to handle Set_Font (textImage) //This handler sets the TextEdit
formatting options to reflect your text image.
Click (textImage.TextFont) //Each text attribute has a corresponding
image.
Click (textImage.TextSize)
(*Include other attributes as needed *)
end Set_Font
```

## Using Marked-Up Text: Advanced

Depending on your TIG, you can use various text markups in your Text Images, shown in the table below.

### Text Image Mark-ups

Markup	TIG	Result
<code>&amp; amp;</code>	All	Generates an ampersand character.
<code>&amp;lt;</code>	All	Generates a less-than character.
<code>&amp;gt;</code>	All	Generates a greater-than character.
<code>&lt;u&gt;, &lt;/u&gt;</code>	Windows TIG	Underlines the marked-up text

### Turning markups on and off globally

The **DefaultUseMarkup** global property determines whether or not Text Images recognize markups by default.

- **Yes:** If the **DefaultUseMarkup** is set to *yes*, all of your TIGs recognize supported text markups by default.
- **No:** If the **DefaultUseMarkup** is set to *no*, markups are treated as literal text.

The default value of the **DefaultUseMarkup** is *no*.

### Using markups in individual text Images

Regardless of your **DefaultUseMarkup** global property value, you can always use (or not use) markups for any single Text Image by including the **UseMarkup** property.

- **Yes:** If the **UseMarkup** property is set to *yes*, the current TIG recognizes supported markups within the Text Image.
- **No:** If the **UseMarkup** property is set to *no*, all markup tags within the Text Image are treated as literal text.

### Markup Examples

(Text: "`<b>This</b> is <i>cool</i> !!", UseMarkup: no)`

Generates: `<b>This</b> is <i>cool</i> !!`

(Text: "`<b>This</b> is <i>cool</i> !!", UseMarkup: yes)`

Generates: **This** is *cool*!!

## Finding Images

### Finding an Image with several possible states

When you are scripting, you can't always count on images on the SUT to stay the same after you capture them. Many objects change color when they are clicked, like a selected icon, or highlighted text. Some change their appearance entirely, like the Live Mode/Capture Mode button in the EggPlant Viewer window.

Since EggPlant relies on image matching to find objects in the SUT, sometimes you have to supply multiple images that represent potential image states.

### Avoiding timing problems

Sometimes a script which has run successfully a number of times suddenly reports that it is unable to find an image in the Viewer window. More often than not, this is a timing issue, rather than a problem with image recognition.

There is a quick test to determine whether or not it is a timing problem: select the failed line and click the Run Selection button in the Script Editor. If the line runs successfully, then it is likely that EggPlant tried to search for the image before it became visible on the screen. This is a timing issue. (If the image is still not found, examine the Viewer window for changes in the image, or try changing the search type of your saved image.)

The following tips can help you prevent timing problems:

### The WaitFor Command

The `WaitFor` command actually holds up the next line of the script until the given image is found. This is particularly helpful when the preceding step opens a new application or web page. (It is not usually needed when you open other parts of an application that is already open.)

#### Example: Using the `WaitFor` command

```
Click "SaveAs"  
WaitFor 5, "SaveDialog"           // Waits for up to 5 seconds for SaveDia-  
                                  log.  
TypeText "FileName"
```

In this example, the script waits up to five seconds for the Save dialog to appear before attempting to type in it. (If the Save dialog does not appear within five seconds, the script fails.)

Be generous in the amount of time you allow for a `WaitFor`; you don't need to try and guess exactly how long a wait you need. Since the script proceeds as soon as the image is found, you don't lose any time on a success. A good rule of thumb is to set the `MaxWait` time for as long as you think a reasonable user would wait; if that is not enough time, you have probably found a problem in your application.

## WaitFor v. Wait

The `Wait` command pauses the script for length of time you specify in the time parameter: (A number by itself represents seconds; you can also type the words *minutes* and *milliseconds*.)

During the `Wait` command, EggPlant does nothing. This is useful if you just want to pause the script at a certain point, but if you are waiting for something to happen on the SUT, you can get more reliable (and potentially faster) results with the `WaitFor` command.

When you use the `WaitFor` command, the time you specify is a *maximum* time, not an absolute time. If the image appears before the maximum time, the script continues immediately.

### Examples: Comparing Wait and WaitFor commands

```
Wait 8 // Pauses the script for 8 seconds.
WaitFor 8.0, "OK Button" // Pauses the script until the image is found, up to 8 seconds.
```

## Remote Work Interval

If you are having frequent, random timing problems with a SUT, you might need to slow EggPlant down a little. You can do this by increasing the Remote Work Interval, the time EggPlant waits between commands sent to the SUT. There are two ways to do this: change the Remote Work Interval in Run Option preferences, or set the `RemoteWorkInterval` global property.

### Remote Work Interval preference

If *most* of your SUTs are having frequent timing problems, change the default Remote Work Interval in Run Option preferences.

- 1 Select Preferences > Run Option > System.
- 2 Increase the Remote Work Interval in small amounts- one or two tenths of a second may be all the difference you need.

### The RemoteWorkInterval global property

If you only need to slow down the SUT in particular scripts (or parts of scripts) try setting the `RemoteWorkInterval` global property on a case-by-case basis.

#### Examples: Changing the RemoteWorkInterval

```
add .1 to the remoteWorkInterval
subtract .1 from the remoteWorkInterval
setOption remoteWorkInterval, .1
```

To write a handler that changes a global property as needed and then changes it back to its starting value, see the [FastImageFound](#) example in [Optimizing Script Performance](#).

## Image Search Time

Image Search Time is the least amount of time EggPlant spends searching for an image. You can change it “permanently” in Run Option preferences, or on a case-by-case basis within a script.

### Note: More about Image Search Time

When you change the Image Search Time, the Search Count and Search Delay values are also changed, and vice versa. Image Search time is always equal to one less than the product of the Search Count and Search Delay values.

## Image Search Time preference

If your Image Search Time is *consistently* too low, you can change the default value in Run Option preferences. (A side effect of increasing the Image Search Time as a preference is that conditional blocks take a lot longer.)

- 1 Select Preferences > Run Option > Screen.
- 2 Increase the Image Search Time in small increments- no more than a half-second.

## The ImageSearchTime global property

If you only need to increase the Image Search Time temporarily (which is usually the best option), you can set the ImageSearchTime global property as needed.

### Example: Changing ImageSearchTime() temporarily

```
put getOption (ImageSearchTime) //Stores the starting ImageSearchTime value.
  into IST                      ue.
set the ImageSearchTime to 2    //Changes the ImageSearchTime value.
(*Proceed with the script here, then...*)
setOption ImageSearchTime, IST //Restores the starting ImageSearchTime
                               value.
```

## Streamlining with Image Functions

This article describes some of the ways you can use image functions to save search time and gather more information about your script execution.

### FoundImageLocation()

The `FoundImageLocation()` function can be a great time-saver when you are calling the same image twice in a row. `FoundImageLocation()` returns the screen coordinates of the last image that was found, so the next time you call that image, you can call it by its screen coordinates and save the time of having to search for it.

#### Example: Using FoundImageLocation

```
WaitFor 8, "Save Button", "OK      //Waits up to 8 seconds for one of the
      Button", "Retry           button images to be found
      Button"

Click FoundImageLocation()        //Clicks the Hot Spot of whichever image
                                was found.
```

In the example above, EggPlant finds one of the buttons during the `WaitFor` command. Then, already knowing the location of a button, it can click it immediately. If you use the image names instead of the `FoundImageLocation()` function, EggPlant must perform the search all over again with the `Click` command.

### ImageFound()

Sometimes you need to know if an image appears on the screen, and take different actions depending on the result. The `ImageFound()` function returns a true or false value indicating whether or not an image (or one of several images) is found.

#### Example: Calling ImageFound() in a conditional block

```
If ImageFound (10, "Save Button") //If "SaveButton" is found, then...
  then
  Click FoundImageLocation()      //Clicks the location of the found
  image.
else                               //Otherwise...
  LogError "Save Button not found" //Logs an Error.
end if
```

### ImageLocation()

If you want to know the location of an image, you can call the `ImageLocation()` function to return the coordinates of its Hot Spot relative to the upper-left corner of the screen. The following example uses

`ImageLocation()` to compare the x coordinates of two images:

### Example: Using `ImageLocation` to compare screen position

```
If item 1 of ImageLocation ("ImageOne") //If the x coordinate of ImageOne
  < item 1 of ImageLocation           < the x coordinate of ImageTwo,
  ("ImageTwo") then                   then...

  Log "Image One is to the left of    //Log a message.
  ImageTwo"
```

**End If**

**Note:** The `ImageLocation()` function raises an exception if the given image is not found.

## Resizing Windows with `ImageLocation()`

Performing a Drag and Drop sequence on the corner of a window could be very complicated if you had to capture an image of the place where you wanted to stop dragging. (What if you only had empty desktop as a reference point?) Fortunately, you can do it by adding to the coordinates of the `ImageLocation()` function.

### Example: Resizing a window with `ImageLocation()`

```
Drag imagelocation("Corner"), //Clicks and holds at "Corner";
Drop imageLocation("Corner") + (10, //Moves 10 pixels to the right and 20
  -20)                          pixels up, and releases.
```

## Searching Part of the Screen

When identical images appear in more than one place on the screen, you often need a way to specify which one you need. One way to do this is to set the `SearchRectangle` global property.

`SetSearchRectangle` takes two pairs of screen coordinates as parameters; these points define two diagonal corners of the search area. (The top-left corner of the screen is 0,0.)

The purpose of the sample script shown below is to click the Customize button (crossed hammer and wrench) in the active window. Since the Customize button in the active window is identical to the Customize buttons in other windows, there is no guarantee that a “Customize” image you find is the right one; however, the red Close button in the top-left corner of the active window is unique. (It is colorless in background windows.) If you make the red Close button the top-left corner of your search rectangle, EggPlant (searching left-to-right, top-to-bottom) surely finds the Customize button on the same window first.

### Example: Adjusting the search rectangle

```
put ImageLocation("CloseButton") into
  UpperLeft

put RemoteScreenSize() into LowerRight
```

### Example: Adjusting the search rectangle

```
Set the SearchRectangle to(UpperLeft,  
    LowerRight)  
  
click "Customize"  
  
Set the SearchRectangle to ( )           //Restores the search rectangle  
                                         to the full SUT screen.
```

## Optimizing Script Performance

By default, EggPlant searches for each image up to six times (with a delay before each search), then finally performs a full screen refresh and one more search before it reports that the image was not found.

If you know at a certain point in your script that an image is already there, or it is not going to be, there is no need to wait for a long search.

Here is an example function that *quickly* looks for an image from a list of passed in parameters:

### Example: FastImageFound

```
function fastImageFound           //Names the function.  
  
set originalSearchCount to the    //Stores the starting value of the ImageS-  
    imageSearchCount              earchCount global property.  
  
set the imageSearchCount to 1     //Changes ImageSearchCount to 1.  
  
set returnValue to ImageFound    //Sets returnValue to the value of the Im-  
    (parameterList)              ageFound function.  
  
set the imageSearchCount to      //Restores the original ImageSearchCount.  
    originalSearchCount  
  
return returnValue               //Returns value of ImageFound.  
  
end function
```

### Adapting this function

Instead of the FastImageFound function, you can insert commands or functions to fit your particular needs. For example, you could make it a “FastClick”.

## Re-using Code

There's no way to get around it: repetition is a fact of life in testing. Fortunately, there are easy ways to let scripts and handlers do the repeating for you.

### Scripts

For code that you can use in several scripts, create a script that you can call as needed. If you use a simple name (with no spaces or special characters) in the same suite or a helper, you can use the script name as a Run command, followed by any parameters it requires.

For a script in an unrelated suite, or a script with spaces or special characters in its name, you can type the Run command, followed by the script pathname.

For a script outside your suite, call the Run command with the script pathname.

#### Examples: Calling scripts from within a script

```
CycleWindows //Calls a simple script name as
                a command.
ConnectionScript "Old SUT", "New SUT" //Calls a simple script name
                with parameters
Run "/Users/Seiji/Documents/EggPlant //Calls the Run command to open
      Suites/Validation.suite/Scripts/ a script from an unrelated
      ConnectionScript" suite.
```

### Handlers

To reuse code within the same script, you can also write a “handler”, which is like a little subscript. A handler is a part of your main script that you can call anytime, as if you were calling another script.

#### Example: CaptureTheScreen handler

```
to CaptureTheScreen prefix, count //Starts the handler.
put prefix & "ScreenShot" & count //Puts prefixScreenShotCount into
    into fileName fileName.
put "/tmp/" & fileName into tempFile //Puts "/tmp/filename" into tempFile
CaptureScreen tempFile //Captures a screenshot
return fileName //Sets fileName as the function re-
    turn value
end CaptureTheScreen //Ends the handler.
```

## Returning Results of a Handler

If a script or handler returns a value, like this one, you can access the value by calling the Results function in the next line. So, using the example above, you can call the handler and access the result as shown below.

### Example: Returning results of a handler

```

CaptureTheScreen "MacTest", 6 //Calls the CaptureTheScreen handler.
put the result into newFile //Puts the returned value into a variable.

```

## Handlers as Commands or Functions

Because the CaptureTheScreen handler is a generic handler, it could also be called as a function. The major difference in use is that a command is a complete statement by itself, and may be used without regard for whether it returns a value, while a function must be called as part of an expression.

### Examples: Calling handlers as commands and functions

```

CaptureTheScreen "MacTest", 6 //Called as a command; runs the handler.
put CaptureTheScreen //Called as a function; returns the value
    ("MacTest", 6) of the handler. (Note parentheses.)

```

## Calling a handler from another script

There are three ways to call a handler from another script:

- 1 Call the script name with a possessive 's, followed by the name of the handler and its parameters.
- 2 Call the script name, adding a dot and the name of the handler to it.
- 3 Call the handler name, followed by the word *of* and the script name.

### Examples: Calling a handler from another script

```

run Logging's CaptureTheScreen "MacTest", //Calls handler with script name
    6 and possessive 's.
run Logging.CaptureTheScreen "MacTest", //Calls handler in the form
    6 script.handler.
run CaptureTheScreen of Logging //Calls handler with handler "of"
    "MacTest", 6 script.

```

## Testing Cross-Platform Applications

The articles in this section explain how you can build upon your current scripts to expand your testing to multiple operating systems.

### Creating platform and language-specific suites

At first, it might seem easy to create a whole new set of scripts for every platform you need to test, but keeping those scripts consistent as your application evolves can be a real challenge.

A friendlier long-term solution is to start with a base script, and call an outside script for each mechanic that changes between operating systems. You can write the script differently for each platform-specific suite, and then tell the script which suite to use for any given run.

#### Example: Quit scripts

in a Windows suite

in a Mac OS X suite

```
TypeText ControlKey & AltKey & "x" TypeText CommandKey & "q"
```

In this example, the script name *Quit* could be called as a command in your base script. Whether it runs the Windows version or the Mac version just depends on which suite you designate.

### Designating the correct suite

The `InitialSuites` global property tells a script which suite (or suites) to check first when it is looking for script or image resources. (These suites even precede the suite in which the current script is running.) If you name more than one initial suite, remember that each suite's helpers (and their helpers) are searched before the next "top level" suite.

In the example above, you could be sure that your base script would find the correct *Quit* script by setting the `InitialSuites` value to the appropriate suite.

#### Example: the InitialSuites

```
Set the InitialSuites to //Names the Windows suite as the first
  ("Windows")           suite to check for resources.
Quit                    //Runs the first Quit script found.
```

**Note:** Like all global properties, the `InitialSuites` can be changed at any point during a script. (*Initial* refers to the initial suite that is searched, not the initial state of the script.)

### Using multilingual text Images

Whenever an interface element can be identified by its text alone, you can use a generated Text Image rather than a captured image in your script. This can save a lot of time creating multiple language suites.

For example, suppose you have to click the same Open button in an application with English, French, and Spanish

versions. You could always capture a separate image of the button for each language, but it would be faster to generate a single Text Image and adjust the text as needed.

### Example: using language-specific Text Images

```
Click (text: "Open", textStyle: "Button")
Click (text: "Ouvrir", textStyle: "Button")
Click (text: "Abierto", textStyle: "Button")
```

## One step further- translation scripts

You can use your language-specific suites even more efficiently by putting a translation script in each one, as shown below.

### Example: Translate script

```
params EnglishWord //Takes an English word...
set translation to {Hello: //Sets the translation variable to a
    "Bonjour", Open: "Ouvrir", Yes: property list, in which each Eng-
    "Oui"} //lish key is assigned a French val-
//Returns the French value of the
return translation's (EnglishWord) English key that was passed in.
```

Now, to generate a Text Image in your base script, you could set the Text value to be the returned value of the “Translate” script.

### Example: Calling the translate script

```
Set the InitialSuites to //Ensures that the "Translate" script comes
    ("French") //from the French suite.
Click (text: Translate //Runs the "Translate" script with "Hello"
    ("Hello")) //as the EnglishWord parameter; uses the
//returned value of "Translate" as the Text
//value in this property list.
```

When you are using a translation script, don't forget to create a version that works for your base language! Even though you don't really need to have your own language translated back to you, your base scripts do need something to do when they encounter the Translate call. The base-to-base “Translate” script does not need to go through

the motions of translating anything; just taking a parameter and returning the same parameter is enough.

### Example: Translate script

```
params BaseWord //Takes a word...
return BaseWord //Returns the same word.
```

## Running from a Master Script

---

The Schedules pane of the Suite Editor has a convenient graphical interface for organizing and running batches of scripts; however, when you need to run scripts more dynamically, you have to write a master script.

The most powerful feature of a master script is the `RunWithNewResults` command, with which you can build upon returned script results to generate the future schedule.

`RunWithNewResults` works by taking another script as a parameter. The parameter script generates its own results, and returns them to the master script. The benefit of this is that you can schedule script runs *conditionally*, based on the return values of previous executions. You can run any number of scripts through a master script, and manage the results as well.

The following example script highlights the functionality of master scripts. First, it runs an initial test. If that test fails, it sends an e-mail warning to a system administrator; otherwise it proceeds to execute a series of tests, storing the results in a text string that it logs at the end.

### Example: Master Script

```

set TestList to ("Test1", "Test2",
    "Test3") //Creates a series of script execu-
            //tions.

RunWithNewResults "InitialTest"

put the result into Outcome

if the status of Outcome is not
    "Success" then //If the result of "InitialTest" is
                    //not "Success", gets the date and
                    //time of the run...

    convert Outcome's runDate to date
        and long time

    sendMail (to: "administrator@
        yourcompany.com", from:
        "EggPlant@yourcompany.
        com", subject: "Initial Test
        Failed", body: "Test run at"
        && rundeate of Outcome &&
        "had" && errors of Outcome &&
        "errors") //Sends e-mail to report the date
                    //and errors of the execution. (&&
                    //joins text strings with a space be-
                    //tween them.)

else //Otherwise...

    repeat with each testScript of //For every script in TestList,
        TestList

        RunWithNewResults testScript //runs the script, and puts the re-
                                        //sults into Outcome.

        put the result into Outcome

        put testScript & ":" && //Adds "Script: Status", then a re-
            status of Outcome && Return //turn character to currentReport.
            after currentReport

        if the status of Outcome is //If the status property is Fail-
            "Failure" then //ure...

            run "CleanupScript" //runs CleanupScript ]

        end if

    end repeat //Ends after the final test in Tes-
                //tList.

    Log "Final Results" //Logs "Final Results"

    repeat with each line of
        currentReport

        log it //Logs each line in currentReport.

    end repeat

end if

```

## Gathering and Using Data

### Reading text on the SUT

---

The ReadText and ReadTable functions use EggPlant's OCR engine to read and return the text in a given area on the SUT. The area can be within a rectangle (determined by two diagonal points), or near a single point.

For simplicity, the following articles refer to the ReadText () function; but the information applies to ReadTable () as well.

### Reading text within a rectangle

When you call the ReadText function with a rectangle parameter, the rectangle absolutely limits the returned text value. That is, if a line of text extends beyond the edges of the text rectangle, the overflow text is not returned.

Like other rectangles in EggPlant, the text rectangle is determined by two diagonal points. Each point can be given as an image name (which represents the location where that image is found) or any other coordinate value.

**Example:** Set `address` to `ReadText ("AddressField", "AddressField" + (20,60))`  
// Sets the address variable to the text in the given rectangle.  
The diagonal points of the rectangle are the location of the AddressField image, and (20,60) past that image.

### Reading text near a point

When you call the ReadText function with a single point parameter, the OCR engine attempts to find a line of text that includes your point, or a line of text that begins near that point.

The point can be given as an image name (which represents the location where that image is found) or any other coordinate value.

### Using the SUT clipboard

---

If the text you want to read can be selected and copied, you can copy the text to the SUT clipboard then return the clipboard contents through the RemoteClipboard() function.

#### Example: Returning text from the SUT clipboard

```
Click "SomeTextField"  
TypeText CommandKey, "a" //Selects All in the text field.  
TypeText CommandKey, "c" //Copies text to the clipboard.  
put RemoteClipboard(5) //Waits up to 5 seconds to return the  
clipboard content generated by previous  
remote command; shows that content.
```

**Note:** On Mac OS X systems, VNC servers running outside of any user account (system servers) cannot transfer clipboard contents. If you want to have access to the SUT clipboard, make sure that you are connected to a VNC server *within* the active user account.

## Data-driven Testing

One of the great benefits of test automation is the ability to run tests repeatedly using different data values. One common approach is to store the data in a text file, then read values from that file during the course of a script. The chunk expressions and direct file access in SenseTalk make this a straightforward task with a formatted text file.

For example, to test a calculator application, you could pass in a text file with number values separated by commas.

### Example: Calculator Test Data File

```
1, 2, 2
3, 4, 12
5, 6, 30
```

The following script drives the calculator to multiply the first two values on each line of the file. Then, it returns the products and validates them by comparing them to the third values.

### Example: Passing in a Text File

```
repeat with theData = each line of file
    "CalculatorData.txt"

    TypeText item 1 of theData //Enters 1st value from the cur-
        rent line of the data file

    Click "Multiply Button"

    TypeText item 2 of theData //Enters 2nd value from the cur-
        rent line of the data file

    Click "Equals Button"

    Click "OutputField"

    TypeText CommandKey, "a" //Selects All in the output field
    TypeText CommandKey, "c" //Copies text to the SUT clipboard
    put remoteClipboard() into Answer //Puts clipboard contents into a
        variable

    if (Answer is not equal to item 3 of //Compares results with 3rd value
        theData) then in the data file

        LogError "Got: " & theAnswer & //Logs discrepancy as an error:
            ", " & item 1 of theData & " x "
            & item 2 of theData & "should be
            " & item 3 of theData
```

### Example: Passing in a Text File

```
end if
end repeat
```

## Creating a Data File

The data you use in your scripts can come from a spreadsheet, as well as a text file. Spreadsheet programs usually have an option to export a file in CSV (comma-separated values) format, or as tab-delimited text.

You can also pass the data directly in a script, as shown below.

### Example: Creating a Data File

```
(* This script creates a test data file with three numbers on each line. The
   third number is the product of the first two. *)
set file "/tmp/CalculatorData.txt" to //Creates a CalculatorData.txt file
  {{                                  in your /tmp directory. (Store
                                  the file elsewhere to keep it
                                  permanently.)
1,2,2
3,4,12
5,6,30
  }}
```

**Note:** SenseTalk makes it easy to read or write the entire contents of a file by using the word `file`, followed by the filename. (See the filename in the example above.) To be sure you're working with the right file, it is a good idea to use the file's full pathname. (For more information, see the section "Working with Files and File Systems" in the *SenseTalk Reference Manual*.)

## Reading and Validating a Data File

Here is an example of a script that reads a returned data file.

### Example: Reading Data

```
repeat with theData = each line of file "/tmp/ //Assigns the contents of
  CalculatorData.txt"                          each line to theData.txt in
                                              turn.
  put repeatIndex() & ": " & theData          //Counts the number of times
                                              the loop is repeated; writes
                                              this value before each line.
```

## Example: Reading Data

```
end repeat
```

If you run the script at this point, Data.txt is returned like this:

```
1: 1,2,2
2: 3,4,12
3: 5,6,30
```

Now that the script can read the data file, the next step is to verify that the values on each line contain useful information.

## Example: Validating Data

```
put zero into goodCount
put zero into badCount
repeat with theData = each line of //For each line of the /tmp/ file...
    file "/tmp/CalculatorData.
    txt"
    put item 1 of theData into num1 //[] Puts the three numbers from each
    line into three different variables.
    put item 2 of theData into num2
    put item 3 of theData into product
    if product = num1 * num2 then //Compares the 3rd number to the prod-
    uct of the first two...
        add 1 to GoodCount //then increments GoodCount if they are
        equal;
    else
        put "Bad Data at line " & //If they are not equal, displays a
        repeatIndex() & ": " & "Bad data" message,
        theData
        add 1 to BadCount //and increments BadCount
    end if
end repeat
put "Good data lines: " & GoodCount //Returns the GoodCount value
put "Bad data lines: " & BadCount //Returns the BadCount value
```

**Note:** The terms *line* and *item* in this script are SenseTalk "chunk expressions" that refer to portions of text. Items are normally delimited by commas, but you can specify any other characters as delimiters by setting the `item-Delimiter` global property. (For more information on chunk expressions and the use of different delimiters, see "Chunk Expressions" in the *SenseTalk Reference Manual*.)

## Timing Script Events

---

This section defines some of the functions you can use to time script events, and provides an example of how you can use them in script logging. (For more information, see the “Working with Dates and Times” section in the *SenseTalk Reference Manual*.)

- **The Date:** Returns the current date
- **The Time:** Returns the current time of day
- **The Seconds:** Returns the whole number of seconds since January 1, 2001

### Example: Time-logging script

```
log "Starting timed task at" && the time && "on" && // Logs the script
    the date                                     start time and date.
put the time into startTime
(* put any code here that you want to time *)
put the time into stopTime
log "That took" && stopTime - startTime && "seconds // Logs the total time
    to complete."                               of the script execu-
                                                tion.
```

Output:

```
2002-07-16 14:33:36 -0600log Starting timed task at 02:33 PM on 07/16/02
2002-07-16 14:33:39 -0600log That took 2.500326 seconds to complete.
```

## Reading Results

EggPlant records a lot of different forms of run data, such as suite statistics, run history, and log files. This section describes these different kinds of data and suggests ways to use them.

### LogFile.txt

---

A LogFile.txt file is a plain text file that contains the detailed log for a script run, with a line for each Log entry. Log entries include the following tab-separated fields:

- **Date and Time.** The exact time of this log entry.
- **Event Name.** The name of the command or function executed, or event that occurred. In the last line, this value is *Success* or *Failure*.
- **Image Name.** The name of the image used for this event; or the text that was typed.
- **Location.** The screen location where image Hot Spot was found; or other information.
- **Line Number.** The line number within a handler.
- **Handler Name.** The name of the handler which produced this log entry.
- **Script Name.** The path name of the script containing the handler.

### RunHistory.csv

---

A RunHistory.csv file is a comma-separated text file that summarizes all of the executions of a script, with one line for each execution. The first line of this file identifies each of the fields:

- **Date Run.** The date and time of the run.
- **Status.** The status of the run.
- **Time.** The elapsed time (duration) of the run.
- **# of Errors.** The number of errors that occurred during the run.
- **# of Warnings.** The number of warnings that were logged during the run.
- **# of Exceptions.** The number of exceptions that occurred during the run.
- **Log File .** The full path to the LogFile.txt file for the run.
- **Return Value.** Any value returned by the script that was run.
- **Error Message.** The final error message for the run, if it was a failure.

This format can be read by many programs, including spreadsheets such as Excel. The second item on each line (Status) is *Success* or *Failure*, so simply reading the last line of this file can be a simpler way to find out whether the latest run of a script succeeded, rather than digging into the LogFile.txt in the individual run folder.

## SuiteStatistics.csv

---

The SuiteStatistics.csv file is another comma-separated-value file, this time with one line for each script that has generated statistics. The fields on each line are:

- **Script.** The name of the script.
- **Last Status.** Either *Success* or *Failure* for the most recent run of this script.
- **Runs.** The total number of runs of this script.
- **Fails.** The number of runs of this script which resulted in failure.
- **First Run.** The date and time of the first recorded run of this script.
- **Last Run.** The date and time of the most recent run of this script.
- **Avg Time(Success).** The average length of time taken to run the script when it was successful. (Failed runs are not included in this time.)

## Reporting Results

---

The 'CSV' format is very easy to import into a spreadsheet program, such as Excel. This section describes some of the other ways to handle your results.

- **RunWithNewResults and the Result.** See "Example. Master script" to observe a way to use RunWithNewResults and the Result function to write results to a log.
- **The ScriptResults().** This function returns the RunHistory.csv file for any script. The script must be in an open suite, or you must provide the full pathname.
- **SendMail.** Used in conjunction with RunWithNewResults or the scriptResults() function, SendMail can report the results of a test run by e-mail. (See "The SendMail Command" in the EggPlant. Reference Manual.)

## Further Reading

---

Here are some other features that are helpful for reporting results that are documented in the *SenseTalk Reference Manual*:

- **The Merge() function.** This can be used to create sophisticated formatted documents using templates, such as reports in HTML or RTF format.
- **File-reading and writing.** The file-reading and writing capabilities of SenseTalk can be used to write results to files on any file system that is mounted on the EggPlant computer, including network mounted drives.
- **URL access capabilities.** You can use these to access remote files through a web interface.
- **The Open Socket, Read From Socket and Write To Socket commands.** These commands allow a script to communicate directly with another running process on the network (if it has a socket interface.)
- **The Shell() function.** The Shell() function enables a script to run any Unix command, including passing test result values off to other processes.

## Organizing Your Testing

The following planning tips can help use EggPlant more effectively.

### Have a Plan

---

When you start developing a test script, you should already be familiar with the application you are testing, and know exactly what you intend to test. Simply put, you should have a test plan.

If you have conducted manual tests of the software in the past, you may have developed manual test scripts – written sequences of steps that a person can perform to verify that particular parts of the system are functioning correctly. Because EggPlant interacts with software just like a person does, these manual test scripts are an ideal starting point for developing automated test scripts.

If you do not have any formal tests already, you may be wondering where to start. Some people choose to model the entire application, and map out a strategy to systematically exercise every part of it. Others choose to focus on areas of a system that present the highest risk, defining the highest risk as the greatest potential cost of failure or the most likely to contain bugs.

Whatever approach you take, the most productive method is to start small, get some simple tests working, and then build larger and more ambitious tests as you progress. Remember that the least amount of automation is already better than none; and every small, manageable step brings you closer to your ultimate goal.

### Think in Terms of End-User Functionality

---

EggPlant interacts with the SUT like a “virtual user”, making it ideal for testing user-level functionality. Think in terms of the high-level tasks that a user performs with your software: What are the major functions that must be working for end users to get value from your product? How can you test these functions? Depending on how much time you have, go down a level and repeat this process.

Kent Beck presents this concept of testing very well in his book “Extreme Programming Explained” (Addison Wesley Longman, 2000.) Many people want to relate testing coverage to a percentage of the application. Beck makes the point that you should not set goals that you cannot measure. When people say they test 80% of their application, what exactly are they counting? Would 100% testing mean that all valid and invalid data cases were tested, or that every permutation of system options and application options was tested? A more meaningful goal is to test all of the application’s essential functions.

### Be Aware of the Visible Interface

---

Automated testing with EggPlant requires that you be keenly aware of how a real user would go through the steps that you want to test. You always have to keep in mind what the user would see and how the user would interact with what is actually visible on screen.

For example, suppose you are testing these actions: Double-clicking on an icon to launch a web browser, typing a URL in the Address field, and pressing Return to load the page.

If these actions are executed too quickly, the browser might not be open when the text is typed. A human user would know to wait for the browser to open before typing a URL, so your script needs to know it, too. A WaitFor “Address field” command would do the trick.

## Establish a Consistent Interface

---

Most operating systems allow users to customize the interface, at least to some extent. This is very user-friendly, but not so friendly to image-based testing. For example, in Mac OS X, you can select the color that is used to indicate the currently chosen menu item. If you capture images of the menu items highlighted blue and then change the color to gray, EggPlant continues to look for blue and probably fails.

It is best if your SUTs are computers that you control, so they are not customized by other users between test cycles. If this is not the case, try to establish standard visual settings for all of the SUTs you use. You might even create scripts that check your settings and change back any that are different than the standards.

Note: For most operating systems, the default settings are the best to adopt. This can make new SUTs ready to use right out of the box; and even SUTs that have been around the block are likely to use many of the default settings.

For Mac OS X, find the following settings in System preferences:

- Appearance: *Blue* or *graphite*, (In Appearance)
- Font-smoothing (in Appearance)
- Scrollbar arrow positions (in Appearance)
- Color depth (In Displays)
- Solid desktop color (in Desktop & Screen Saver)

For Microsoft Windows systems, find the following settings in the Display Properties control panel:

- “Use the following method to smooth edges of screen fonts”: *Off*
- All settings in the Taskbar and Start Menu Properties control panel
- All settings in the Folder Options control panel.

Regardless of the operating system you are testing, standardize the color depth at which all of your SUTs of that type run. Some older computers cannot display 32-bit color (“millions of colors”) and even if they are able to display 24-bit color, the colors that they send through VNC can be very different from those displayed on a 32-bit system. (Some of these color differences might fall within your search type tolerances, but others can cause a script that was generated on a 32-bit system to fail on a 24-bit system, and vice versa.) If you have SUTs with different maximum color depths, you need to decide whether you are going to use the lower color depth for all of them, or capture an additional set of images if they are not matching up.

Finally, keep in mind that solid desktop backgrounds are easier to use than pictures or patterns. (If you capture an image with part of a picture in the background, EggPlant might not be able to recognize it at another place on the desktop.)

**Note:** Screen size and resolution do not affect image matches.

## Record the System State

---

One of the benefits of testing on multiple SUTs is the ability to observe how an application responds to different system set-ups. Here are some considerations:

- What Operating System is running on the SUT?
- How much memory does it have?
- Which System preferences differ between SUTs?

By documenting these and similar data, you can address each configuration with tests to validate its impact on the application.

## Organizing your Images

---

This article offers practical advice for organizing images and reducing your own image search time.

### Folder structure

Subfolders in your Images directory can reduce clutter quite a bit. It helps to have a theme, such as separate subfolders for each of your applications windows or panels. From there, you can subdivide further, in any groupings that make sense to the people who use your suite.

The Suite Description field in the Settings pane is a great place to explain your folder structure.

### Naming conventions

Naming conventions make it a quite a bit easier to identify your images, and *much* easier to identify your co-workers' images. The table below shows one QA team's image-naming conventions; it might be a helpful starting point for your own guidelines.

## Sample naming conventions

Image attribute	Denoted by...
Highlighted	<i>_h ending</i>
Disabled	<i>_d ending</i>
Normal	No <b>ending</b>
Menu	<i>m_ prefix</i>
Menu item	<i>mi_ prefix</i>
Checkbox	<i>cb_ prefix</i>
Radio button	<i>rb_ prefix</i>
List box	<i>lb_ prefix</i>
Text box	<i>tb_ prefix</i>
Group box	<i>gb_ prefix</i>
Slider	<i>sl_ prefix</i>
Button	<i>bt_ prefix</i>
Pull-down menu	<i>pd_ prefix</i>
Selected pull-down menu item	<i>pds_ prefix</i>
Tab menu image	<i>tm_ prefix</i>
Dialog title	<i>t_ prefix</i>
Screen capture	<i>sc_ image</i>

## Connecting under Special Circumstances

### Direct Connections

If you are testing a SUT that cannot be part of a larger network (usually for security or logistical purposes), this section explains how to create a closed, local connection with EggPlant.

#### Setting up a Direct Connection on Each Machine: Step-by-Step

This process is nearly identical for the EggPlant computer and the SUT. (The only difference is the IP address in step three.)

The required settings are found in the computers' network preferences (not in the EggPlant application or the VNC server.)

- 1 **In network preferences, select (or create a new) connection.** Select a "wired" or "ethernet" connection.
- 2 **Indicate that you are using a static IP address.** This option may take several forms:
  - Disable *DHCP* or *Roaming Mode*.
  - Select *Static* or *Manual* IP address.
  - Select *Use the following IP address*:
- 3 **Enter a private IP address.** Use *10.0.0.1* for EggPlant, and *10.0.0.2* for the SUT. (These IP addresses are "non-routable", or reserved for private networks.)
- 4 **Make the EggPlant computer a router for the SUT.**
  - A. In the *Gateway* or *Router* field, enter the IP address of the Eggplant computer: *10.0.0.1* .
  - B. Enter the subnet mask *255.255.255.0* . (On this network, only IP addresses that begin with 10.0.0 are considered local.)
- 5 **Connect the computers with an ethernet cable.**

### Reverse VNC Connections

If you are unable to configure the firewall of a SUT to accept VNC connections, you can often open a reverse connection, in which the SUT initiates the VNC connection and the EggPlant computer accepts it.

The EggPlant side of reverse connections is described below; for the SUT, please refer to its VNC server documentation.

### **Preparing EggPlant for Reverse Connections: Step-by-Step**

- 1 In EggPlant's VNC preferences, select the *Listen for Reverse Connections* checkbox.
- 2 In the *Port* field, specify the port number for reverse connections. (The standard port for reverse VNC connections is 5500.)

Configure your router and (and any additional firewall on your system) to allow connections on port 5500.

## Troubleshooting Connection Issues

Connection error	Possible Cause	Solution
<i>FAILED: No such host</i>	EggPlant does not recognize the network name you are using.	Try using the SUT's IP address instead.
<i>FAILED: Temporarily unable to connect: Operation timed out</i>	EggPlant cannot see the IP address.	Open the Network Utility and try to ping the IP address. If you can't ping the IP address, then you could be using the wrong IP address.  Make sure that the SUT's firewall is allowing VNC connections.
<i>FAILED: Temporarily unable to connect: Connection refused</i>	The IP connection to the SUT has been established, but EggPlant cannot connect to the VNC server on the SUT.	The IP connection to the SUT has been established, but EggPlant cannot connect to the VNC server on the SUT.
<i>FAILED: Remote Login Failed - Password Rejected</i>	The password in the Connection List is incorrect, or the password in the VNC server was typed incorrectly.	Re-enter the password in the Connection List.  Try retyping the password in the VNC server on the SUT.
Viewer window updates are very slow in Live Mode.	This is probably a network issue.	Discuss the issue with your system administrator. (As a test, create a direct connection between EggPlant and the SUT to see if this improves performance.)

