Extend the Range and Reduce the Costs of Your SAP Testing Activities with eCATT

Jonathan Maidstone

Jonathan Maidstone joined SAP in 1996 as a translator for the ABAP Language and ABAP Workbench departments. Following a spell as a technical writer, during which he was involved with the SAP Control Framework and DCOM Connector projects, he is now a product management specialist with particular responsibility for rolling out eCATT, SAP's new test tool.

(complete bio appears on page 110)

Functional testing is vital to the success of every SAP implementation. However, the resources required to create test cases, assemble and brief a team of testers, and collect and collate test feedback can be considerable, especially if you take into account the correction and retesting phases of the project. Thus an automated test tool, in addition to human testers, is indispensable.

Of the various automated test tools on the market now, most are Windows-based and can be used to test any other Windows-based application — which means that you can use these tools to test an SAP system using the SAP GUI. However, such tools only view the SAP GUI as a Windows application, and not as a single layer of a multiple-tier system. Consequently, these tools can only enter or retrieve the data contained in fields on screens at the frontend. They cannot be used to test backend functions, which we all know can be considerable given the dependencies inherent within the SAP system itself. With a Windows-based test tool, you could not, for example, check that certain pieces of data have been written properly to the database, or cross-check a result with another part of the system.

A popular tool with SAP shops, SAP's Computer Aided Test Tool (CATT), which has been around since R/3 Release 3.0, *can* record these types of backend transactions, assign variables to input and output fields, perform field checks, and create complex test cases that cover all of the transactions in a business process. However, CATT is no testing panacea, and is, in fact, starting to show its age. It *cannot* test the UI

Automated Test Tools for the Uninitiated

While I suspect most readers have some personal experience with automated test tools, for those who do not, I like to draw an analogy between an automated test tool and your everyday VCR: both allow you to make a recording of a part of the system as you work through it. When you have finished recording, the tool converts the recording into commands that form a script. The script describes everything that you entered during the recording (e.g., field values you entered, buttons and function keys you pressed) and also contains information about the system's actions (e.g., output values, error messages displayed, etc.).

Here, however, the VCR analogy ends, for a test script is not a "film" that merely plays back what it records. Instead, it is a dynamic object that you can modify. You can replace the values that you entered with variables, allowing you to test your applications repeatedly with different sets of values ("Will it work if I use sales channel 11 instead of 10?"). Once you have edited your script in this way, you can execute it. The test tool then performs the same actions that you performed (for example, creating a sales order), but it uses the modified data instead of the same values you entered during the recording. The result of the test is then presented to you in a log, which tells you exactly what you entered, what the system did, and what the results were.

controls[1] users clamor for — the controls that make applications easier to navigate and operate, such as the SAP Tree Control, SAP List Viewer, and SAP HTML Viewer — and it cannot test applications running outside of the SAP GUI for Windows and SAP GUI for Java. It is also difficult to run tests across distributed systems with CATT. SAP Web Application Server 6.20 closes these gaps with the new *extended* Computer Aided Test Tool (eCATT).

In this article I will introduce you to eCATT's extended testing capabilities. I will then explain how testing teams can leverage one of the most sought-after of its new functions — the ability to test transactions that use controls. Mastering this capability will help you expand the range and reduce the costs of your SAP testing activities.

And even if you have no plans to switch from CATT to eCATT at the present time, SAP Web Application Server 6.20 is the underlying technology platform for R/3 Enterprise Release 4.7, and will also be used as the technology platform for future releases of other mySAP solutions, such as the SAP Solution Manager (Release 2.2 or higher). So even if you decide not to use eCATT right now, it will be an integral part of future releases of your SAP systems and, like CATT, will be delivered as part of the Web Application Server at no additional cost! Reading this article will help you make an informed decision about using this new technology when the time comes.

Let's start with a quick review of the existing capabilities provided by CATT and the ways in which eCATT builds on and adds to these capabilities.

Treading New Testing Territory with eCATT

If you are currently using CATT, you know that

[1] Implemented as OCX components in the SAP GUI for Windows and as JavaBeans in the SAP GUI for Java.

this tool offers the ability to test the following kinds of operations:

- **Transactions:** The main focus of CATT is testing SAP transactions. You can record a transaction, assign variables to its input and output fields, and perform field checks — for example, to check whether the default currency you set in your customizing settings is really the one that appears next to the "Amount" field in the transaction display.

- **Table operations:** CATT also allows you to check whether data has been written to (or deleted from) the database properly. It allows you to check customizing settings and simulate them, so that you can analyze the effects of a potential change before making that change.

- **Function modules:** Sometimes within a test it is useful to be able to call a function module — for example, one of the calendar function modules to find out on what day of the week a particular date falls. You can use the results of the function module call as importing parameters of another transaction.

- **Distributed business processes:** In the past, many SAP installations ran on a single R/3 installation, making testing easy since all of the business processes ran on the same box. Nowadays, a complex mySAP.com installation might also include components such as CRM, APO, SEM, or BW. An end-to-end business process test will therefore need to address all of these systems. Although this is possible in CATT, it is not easy. There are two ways of doing it:

  - *Maintain each test object in the system in which it was intended to be run* — While this method technically works, the difficulty is that it leaves your test team working in a disparate environment, and maintaining scripts in several different systems.

  - *Maintain test objects centrally, but specify an RFC destination using a special CATT variable* — This enables you to run tests from a central system, but the handling is not specific enough to enable really flexible testing.

With the help of SAP Web Application Server 6.20, eCATT builds on the capabilities provided by CATT in the following ways:

- **Transactions:** Like CATT, the main focus of eCATT is to test transactions, and as with CATT, eCATT enables you to record a transaction, assign variables to its input and output fields, and perform field checks. However, eCATT goes a step further and enables you to test:

  - *Transactions that use controls* — The testing capability of CATT does not cover transactions that were redesigned in the EnjoySAP initiative, such as ME21N (Create Purchase Order) in the core R/3 system, or that make extensive use of controls, like the tree control and dropdown list box control. Because of this, during your upgrade to 4.6B or 4.6C you have likely found that CATT does not provide sufficient test coverage. eCATT provides a new command option (called the SAPGUI command) that closes this gap and allows you to test controls-based applications. This new command is the focus of this article.

  - *Transactions running externally* — CATT allows you to test only applications within the SAP GUI for Windows and SAP GUI for Java. However, today's system landscapes contain a whole host of web applications — including many from SAP itself, such as Internet Sales and the web frontend of the Business Information Warehouse (BW) — that you will need to test. eCATT provides an interface that allows manufacturers of third-party test tools to integrate their products with eCATT. You can then use an

external tool right from within eCATT to test web applications and external Windows desktop applications.

- **Table operations:** Like CATT, eCATT enables you to verify that data has been written to (or deleted from) the database properly.

- **Function modules:** As with CATT, eCATT allows you to use function module calls in the test script. However, with eCATT you can use function modules with structured or tabular parameters, like BAPIs, which is not possible with CATT.

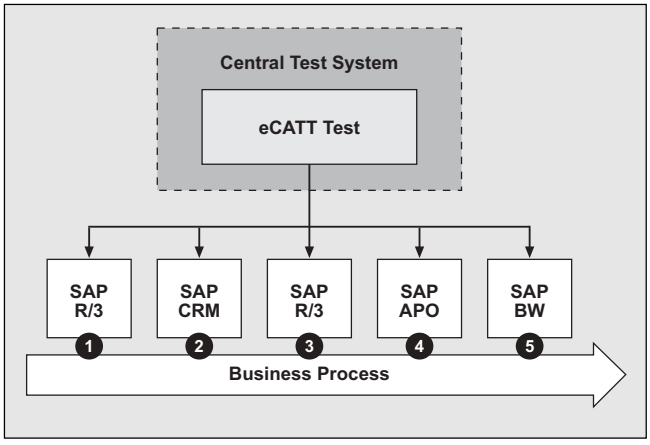
- **"Inline" ABAP routines:** With CATT, if you need to implement any processing that is not covered by CATT commands, you have to write a function module especially for the task. With eCATT, you can write blocks of actual ABAP code within a test script, enabling the full power of ABAP for any special requirements (e.g., testing a complex database query that returns several records from more than one table, but has no function module or BAPI). While powerful, this feature should be used with care, since the more ABAP coding an eCATT test script contains, the harder it can be for someone else to read, understand, and maintain that script.

- **Distributed business processes:** With eCATT, it's as easy to run the same test script on several different systems (see **Figure 1**) as it is to run a test on a local system. From the central eCATT test system, which contains all the testing objects, you can test an order entry in CRM, switch over to R/3 to ensure that the order replication worked, and then test the production planning in APO. Also, the target systems can run on 4.6C[2] or higher. Only the central eCATT system has to run 6.20, so you can set up a standalone 6.20 server devoted to running tests and use eCATT straight away without having to upgrade any of your target systems running on 4.6C.

  In addition, most eCATT commands can take a destination as one of their arguments, allowing you to run that command *and that command only* against a particular system. The next command might then have the same destination, a different destination, or no destination at all (in which case it executes locally in the test system, often useful for utility functions that interpret results returned by a transaction). Contrast this with CATT, where this is only possible by setting the destination implicitly (assigning an RFC destination to a special CATT variable, after which all subsequent calls would be executed against that destination). As you can imagine, implicitly setting destinations in this way is far more susceptible to error than an explicit destination assignment.

As you can see, eCATT provides you with greater testing flexibility than CATT (see **Figure 2** for a side-by-side comparison). Of course, as with most things, this flexibility comes at a price — CATT has only one way of recording transactions; sometimes it works, sometimes (for example, because of the use of controls) it doesn't, but you can't choose the wrong method since there is only one. With eCATT, you

Figure 1 Testing a Distributed Business Process with eCATT



[2] Minimum support package levels apply (see SAP Note 519858 for up-to-date details of the support package level required to run eCATT in 4.6C, 4.6D, and 6.10 systems).

Figure 2 A Comparison of CATT and eCATT Testing Capabilities

Testing Capability	CATT	eCATT	Description and Examples
Transactions that *do not* use controls	✓	✓	Both CATT and eCATT can test transactions that do not use the SAP Control Framework — e.g., FB01 (Post Document) or VA01 (Create Sales Order). If you are familiar with recording transactions in CATT, you will quickly feel at home with this technique in eCATT, as the capability itself is unchanged.
Transactions that *do* use controls		✓	Only eCATT can test transactions that use the SAP Control Framework — e.g., ME21N (Create Purchase Order) or BP (Business Partners). With CATT, it *looks* as though the recording is working, but you would later find that the test script could not address the controls.
Transactions running externally		✓	Only eCATT provides the external tool integration that allows you to test web applications and external Windows desktop applications (including CRM Mobile Sales, for example) running outside of the SAP GUI for Windows and the SAP GUI for Java.
Table operations	✓	✓	CATT and eCATT have a significant advantage over other test tools because they can access the SAP database, and they allow you to read and check entries. For customizing tables, you can also make temporary changes to simulate what would happen if you were to make a certain change. At the end of the test, you can restore the original settings or commit the changes to the database. The capability itself is the same in CATT and eCATT.
Function modules	✓*	✓	Both CATT and eCATT enable you to include function module calls in your test script. However, only eCATT allows you to use function modules with structured or tabular parameters (e.g., BAPIs). Calling a BAPI can be a good way to quickly read a set of data, such as a list of sales orders, that you can then feed into subsequent parts of a test.
"Inline" ABAP routines		✓	eCATT allows you to use actual ABAP coding directly in your eCATT test scripts, enabling you to manipulate data beyond the scope of eCATT's own scripting language.
Distributed business processes	✓*	✓	In CATT, it is possible to run tests across systems, but it is not easy — you must either maintain a test object in each system or specify an RFC destination from a central system using a special CATT variable. With eCATT, you can create a map of your system landscape that contains all of the systems a test script might need to use, and then assign any of these systems to a particular command in the script.
* Possible in CATT, but limited functionality.			

have *three* different techniques (or, as they are called in the eCATT world, "drivers") for recording an application:

- The *TCD command* is the existing CATT recording technique that you are likely already familiar with. A little later on, I will briefly take you through a TCD recording, before comparing this technique to the new SAPGUI command.

- The *SAPGUI command* is the new eCATT recording technique for coping with transactions that use the SAP Control Framework. This new technique is the focus of this article, and I will walk you through it in detail using an example transaction.

- The *interface to external tools* allows you to fully integrate a certified external test tool[3] into the eCATT test environment. The TCD and SAPGUI commands only allow you to test transactions running in the SAP GUI for Windows and SAP GUI for Java, and this technique enables you to test web-based or Windows desktop applications. To do this, you create an eCATT script, but specify a particular external tool for recording and editing instead of using eCATT commands. The script is then created with the external tool, allowing you to leverage that tool's particular strengths. When you save the script, it is recorded in the SAP database and can be called from any other eCATT script.

  The particular steps for using the external tools technique are highly dependent on the external test tool that you choose to use for your test project, so I will not go into the details of this driver in this article.[4]

[3] For updates on certified external tools, check **www.sap.com/partners**.

[4] For more detail on using external test tools with eCATT, refer to the online help for SAP Web Application Server 6.20 at **http://help.sap.com**.

Before we start walking through the steps of testing a transaction, it will help to have an idea of how the different components of the eCATT environment fit together. We'll take a quick look at these basic elements next.

What You Need to Know About the eCATT Environment

At this stage, I want to introduce some of the basic terminology and components of the eCATT world and show you how they fit together, so that you have a good understanding of the "big picture" when we are ready to tackle the actual testing process. These components include test scripts, test data containers, test configurations, and system data containers.

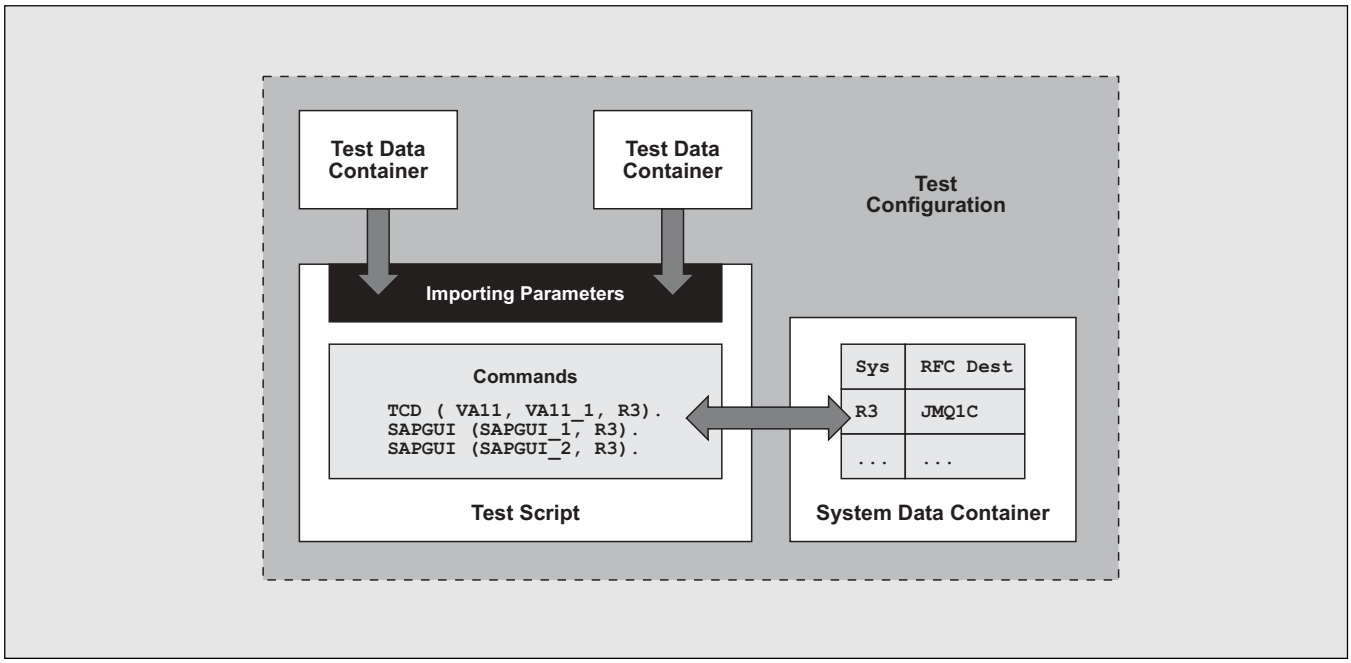
Figure 3 shows an overview of how all these components work together.

Test Scripts

There are various kinds of objects with which you will work in a test project, the most important being the *test script*. A test script comprises the application data that you have recorded (usually one or more SAP transactions), and includes the changes that you make to the input values (by assigning variables instead of fixed values) and any additional functions that you require, such as database checks, plausibility checks (making sure that the planned delivery date of an order isn't yesterday), or retrieval of values from output fields on the screen.

eCATT scripts are written using a scripting language that is similar to ABAP. You can regard this language as an extension of the now-familiar CATT command set. The eCATT scripting language contains not only the commands that you need to record transactions, call function modules, and reference other eCATT scripts, but also provides a host of

Figure 3 The Components of the eCATT Environment



other commands and constructions. For example, the DO command allows you to program loops within a script. This capability allows you, for example, to repeat a recorded transaction any number of times, which is useful if you want to test how a transaction reacts when a certain threshold is exceeded. There is also an IF construction, so you can make parts of your test script execution dependent on what has already happened.

Along with the commands, eCATT offers a set of special variables (similar to the SY- variables in ABAP) that provide information on the current system status (see **http://help.sap.com** for details).

✓ *What Makes a Good Test Script?*

So, what makes a good test script? In my opinion, there are two things that you should bear in mind:

- ***Think reuse!*** *Don't make your scripts too large and unwieldy. By that I mean that you shouldn't try to cover a whole business process in a single script. Instead, break the process down into its components (even right down to the transaction level) and record separate scripts. That way, you will be able to reuse individual scripts in several different test processes (remember that scripts can call other scripts).*
- ***Have a definite result in mind!*** *Recording a transaction is only half of what makes a good test case. You should have a plan for precisely what you want your test to tell you. For example, do you want to know the customer number of the customer you just created, or do you only want to see the success message "Customer created"? For that matter, do you want the application to succeed at all? If you are testing a function that is supposed to have restricted access, an error message (e.g., "No authorization to do this") might be your definition of success!*

Test Data Containers

A test script on its own, however, does not contain all of the information required to perform a complex test. You will, for example, need to supply the script with data. Depending on what exactly you want to test, a typical application test might involve running a transaction several times — once with data that you know the transaction must accept if it is running correctly, and then again with various sets of incorrect data (wrong date formats, dates in the past, removal of more widgets from the warehouse than are actually available, etc.) to see how the transaction reacts to these irregular situations.

To make this data available to more than one script (after all, several different test scenarios might start with the same one or two transactions), eCATT stores the data separately from the test scripts in an object called a *test data container*. This is not the case with CATT, where each test procedure has to have its own data because it is stored with the procedure.

Test Configurations

Since test scripts in eCATT no longer have data directly assigned to them, another kind of object needs to be created once the test script is finished and ready to be used productively.

A *test configuration* links a test script with one or more test data containers and allows you to set up a "real" test in which the script is executed several times automatically, using different sets of data derived from the test data containers.

System Data Containers

Your testing activities generally don't start with test scripts, test data containers, and test configurations, however. If you look back at Figure 1, you will notice that we still have a practical problem to surmount before we can record a script, namely that eCATT usually runs in a different system than the system (or systems) in which the applications you want to test are running. So, the first thing to do at the beginning of a test project is sit down, define the scope of the project (i.e., the applications you want to test), and, based on that, create a description of the system landscape in the form of a *system data container*. (More on the benefits of this approach in the section "Step 1: Map Your System Landscape.")

Let's now see how eCATT can be used to test a controls-based transaction running in a distributed environment.

Testing a Transaction with eCATT

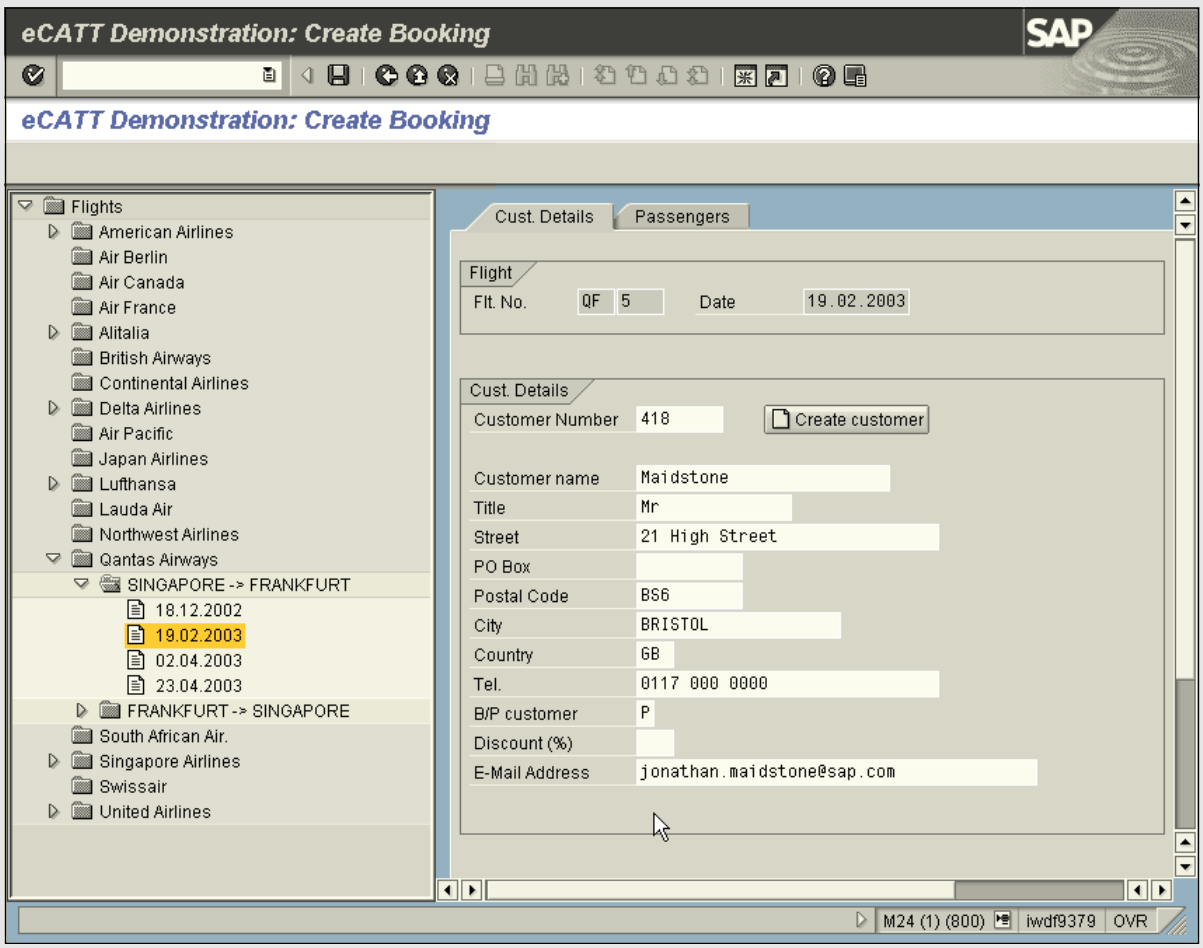
There are four simple steps involved in testing a transaction with eCATT:

1. Map your system landscape.
2. Create a test script.
3. Record the transaction.
4. Execute the test script.

In the next sections, I will show you how to test an example application, running in a distributed environment, that uses the SAP Control Framework — an integral and widely used GUI technology for applications that until now testers had no way of testing automatically.

Before jumping right into the steps, let's first take a look at the example application we will be testing (see **Figure 4**). In good SAP tradition, it is based on the flight data model used for ABAP training and is used to create bookings. On the left side of the screen is a tree control, in which you can select a flight. When the user right-clicks a date in the flight list (e.g., "19.02.2003"), a context menu (also a control!) appears. From the context menu, you can proceed to create a new booking. The required data for the

Figure 4 An Example Flight-Booking Application That Uses Controls



booking is entered in the tabs on the right side of the screen: “Cust. Details” and “Passengers.”

The presence of the tree control and the context menu (as well as an ALV grid control on the “Passengers” tab, which is not visible in Figure 4) makes this transaction a clear candidate for recording using the new eCATT SAPGUI command.

Over the course of the rest of this article, we will walk through the steps for actually recording and testing our example controls-based transaction, beginning with how to map your system landscape.

Step 1: Map Your System Landscape

eCATT, like any other SAP application that needs to communicate with another SAP system, does so using Remote Function Calls (RFCs).

However, instead of *directly* addressing a remote system over an RFC destination, as you would with CATT, with eCATT you create a map of your system landscape in the object I mentioned earlier, a *system data container*. This is what makes it so easy to test transactions in a distributed environment (more on this in a moment). Typically, you should create one system data container per project that describes all of the systems you need to use within the scope of this particular project. You can attach any number of individual test scripts to this one system data container.

To create a system data container (and, for that matter, any other eCATT object, including test scripts, test data containers, and test configurations), log on to your system and start transaction SECATT. From the eCATT initial screen, select the kind of object you want to create (in this case, a system data container), type in a name, and then click "Create."

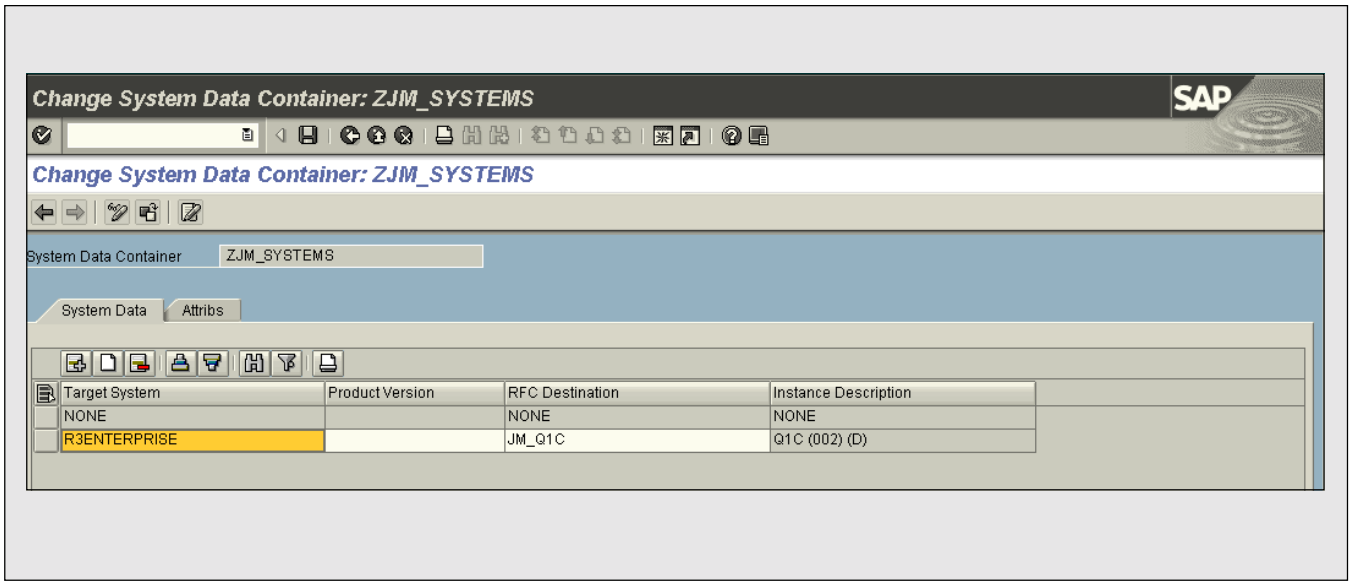
Each entry in a system data container describes the relationship between the central test system and *one* of your target systems. **Figure 5** shows the definition for system data container "ZJM_SYSTEMS" as it appears in the eCATT test system. It contains the following details:

- **Target System:** A logical name (freely definable) that describes the target system. I have used the name "R3ENTERPRISE" in the example. The name you enter is the name you use within a test script when you want to indicate the target system in which it is to be executed.

- **Product Version:** An optional documentary field that allows you to enter details of the system. In Figure 5, this field is empty, but you could enter something like "R/3 Enterprise Test System" to further qualify the description of your target system.

- **RFC Destination:** Here you enter the name of an RFC destination, which must have already been created in transaction SM59 (you can see that I created the destination "JM_Q1C"). This is the very same combination of system, client, language, and user information that eCATT will use to log on to the target system.

Figure 5 The Definition for System Data Container "ZJM_SYSTEMS"



- **Instance Description:** The information on the instance is fetched automatically from SM59 and describes the technical attributes of the RFC destination (system, client, language). The example description "Q1C (002) (D)" indicates target system Q1C, client 002, and language D (German).

In addition to the target systems that you enter into the system data container, the destination "NONE" is always present. "NONE" is a default RFC destination that points to the current client of the current system.

> ✓ ***Tip***
>
> *Once you have defined the scope of a test project, create a single system data container in which all of the target systems required for that project are defined. The fewer system data containers you use, the easier it will be to manage your test landscape.*

You may be wondering at this point why system data containers are necessary in the first place — after all, the RFC destination created in transaction SM59 already provides the required information about the target system, so why complicate the issue with this "target system" business? The answer is, in a single word, *flexibility*.

Suppose you want to run a test script in two different system landscapes — for example, in a customizing test client, and then in a training system to ensure that your training environment is set up correctly. To do this from a central test system, you need two RFC destinations: one pointing to the test client and the other pointing to the training system.

If you were to use the RFC destinations to directly specify the target of the script, you would have to *change the actual script coding* in order to switch between the target systems, as is the case with CATT.

Now consider how the whole thing works if you use system data containers. You can have two system data containers, each of which contains an entry with the target system name "R3ENTERPRISE." However, the RFC destination attached to this target system name is different in each container. In your script, the target is always "R3ENTERPRISE," but by changing between the system data containers, you can execute the same script against different systems, this time *without having to change the coding of the script itself*.

> ✓ ***Tip***
>
> *RFC destinations are security-critical! In order to achieve unattended testing, eCATT must be capable of logging on to a target system without needing a user to enter a password. The most secure way to do this is to set up a trusted RFC relationship between your source and target systems.*

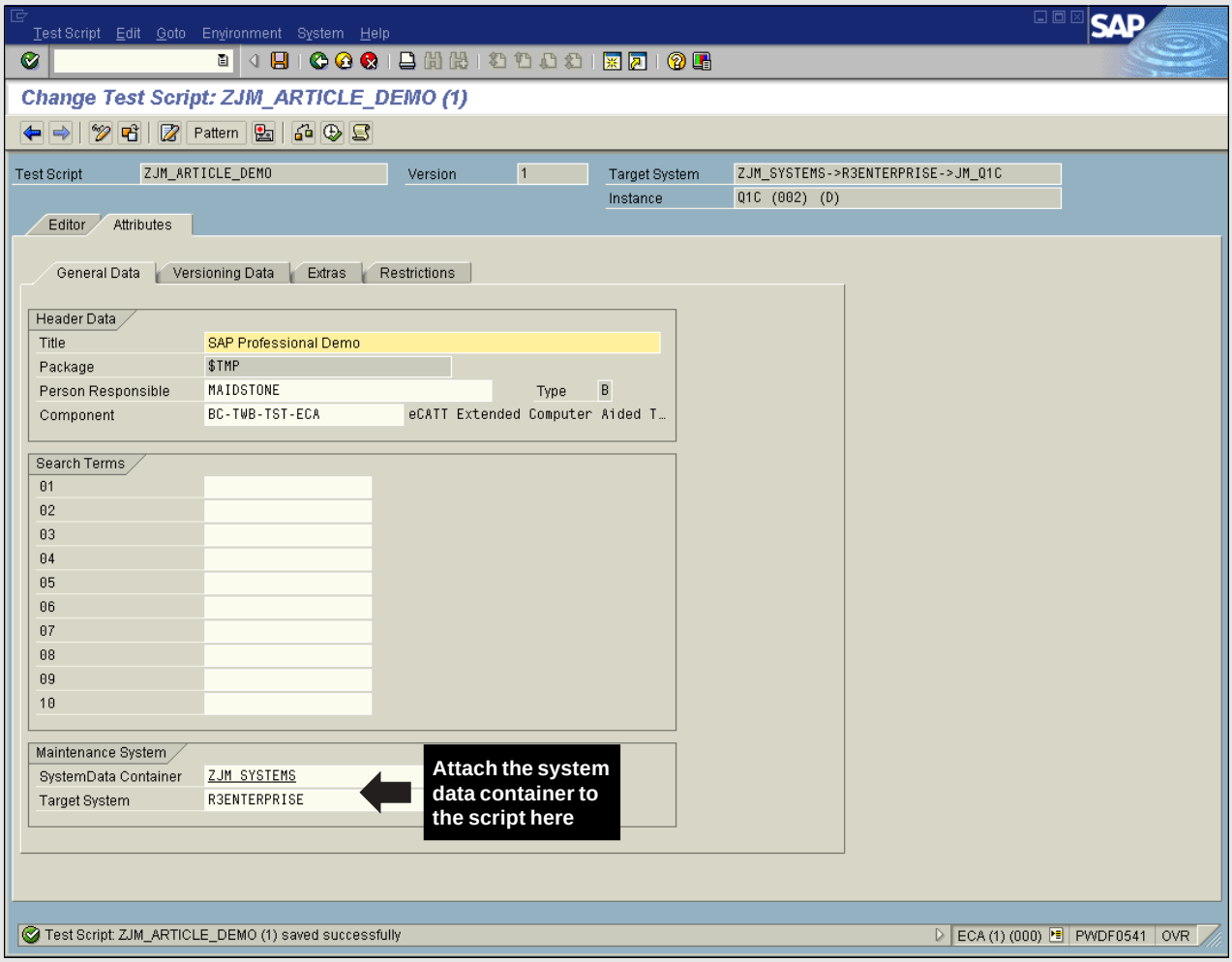
Once you have mapped out your system landscape using a system data container, you are ready to create a test script. Let's go through the tasks involved in this second step.

Step 2: Create a Test Script

Creating a test script in eCATT is not significantly different from creating a test script in CATT, so the tasks involved in this step are likely very familiar to most of you, and I will only cover them briefly here.

To create a test script, on the initial screen of transaction SECATT, select the "Script" option. Enter a name ("ZJM_ARTICLE_DEMO" in the

Figure 6 Maintaining Test Script Attributes



example) and choose the "Create" icon. This brings you to the attribute maintenance screen for the new test script (see **Figure 6**).

Here, you must assign a title ("SAP Professional Demo"), a package ("$TMP"), and an application component (BC-TWB-TST-ECA, i.e., eCATT) to the test script. It is also here that you can attach a system data container to the script. By doing so, you make all of the systems listed in the system data container available to the commands in your script. In Figure 6 you can see that I attached the "ZJM_SYSTEMS" system data container created in Step 1 (refer back to Figure 5).

Once you have created the test script and attached the system data container, you are ready to record the transaction.

Step 3: Record the Transaction

Before you can start recording a particular transaction, you need to decide *how* you will go about recording it.

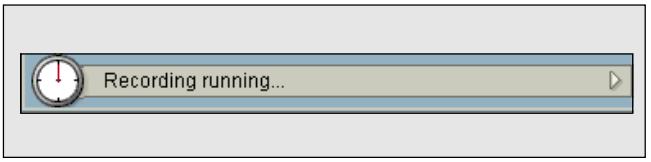
Remember, CATT only has a single recording technique, so picking the wrong driver is not an option. With eCATT, on the other hand, you need to look at the user interface of your transaction and select the best driver for recording, which will be the existing TCD command[5] (the original CATT technique for transactions that do not make extensive use of controls), the new SAPGUI command (for transactions that are full of ALV grid controls, tree controls, and so on), or the interface to an external test tool.

> ✓ ***Tip***
>
> *Before you start recording a transaction, always go through a practice run first. Otherwise you risk picking the wrong driver for the job!*

The best way to ensure that you always pick the right driver for the job at hand is to understand why eCATT has both the TCD and SAPGUI drivers in the first place.

The *TCD driver* uses the batch input technique to record the data from each screen of a transaction. You will already be familiar with this driver if you have used CATT, and you will find that aside from the new eCATT user interface, the features have changed very little. When you select the TCD driver for recording, specify the transaction you want to record. eCATT starts the transaction, and then you proceed as you normally would. The only difference you will notice is that each time you press "Enter," click a pushbutton, or choose a menu entry or function key (actions that cause control to pass from the SAP GUI back to the application server), the message in **Figure 7** will appear in the status bar.

Figure 7 The Status Message Displayed When Recording with the TCD Driver



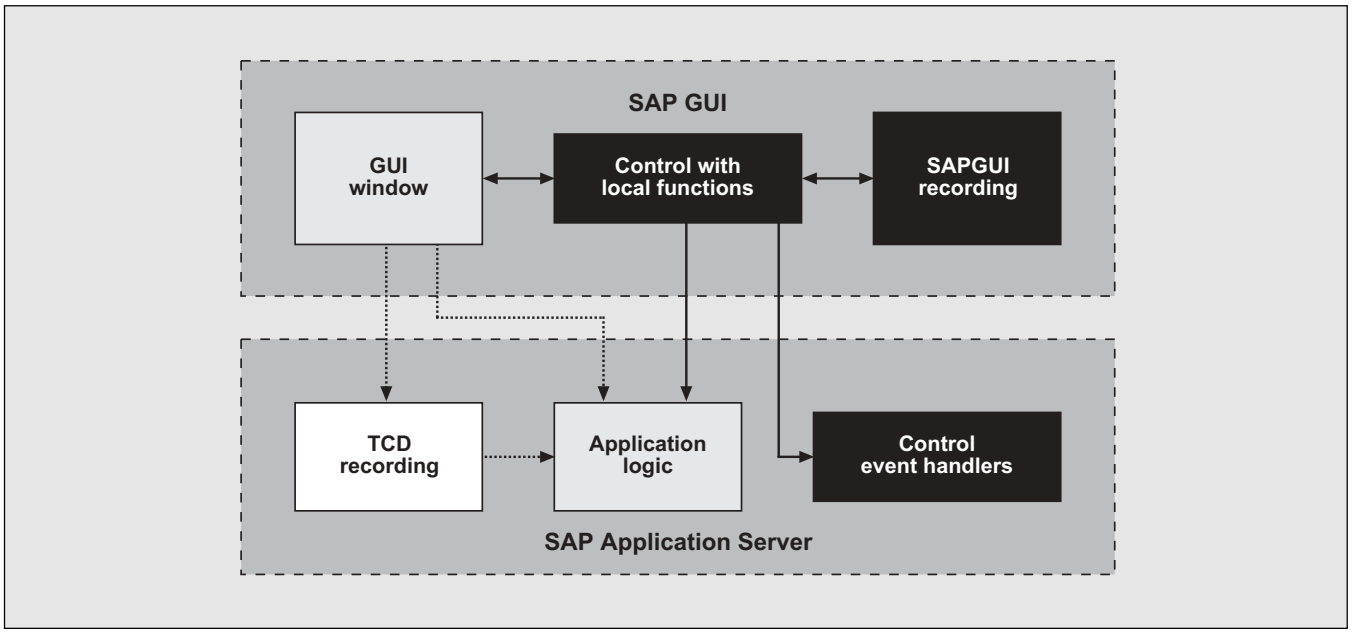
At this point, the system is collecting all of the field contents from the screen, plus the function code (the special code attached to whatever function you triggered that tells the program what to do next). This is the information that eCATT will require to play back the transaction.

The TCD recording mechanism is fairly straightforward. However, it relies on user interface interactions to send their data back to the application server along the same route — and this was the case up to and including Release 4.0B. In subsequent releases, SAP Control Enabling Technology (renamed the SAP Control Framework in Release 4.6B) introduced new UI elements (controls) that do *not* follow the same route. Instead of sending information back to the application server, where functions are processed, these controls also implement some of their functionality locally at the frontend, meaning that a control's state could change without being recorded by the TCD driver. Other control functions are implemented using the object-oriented concept of events and event handlers, which also bypass the transaction recorder.

In short, it is possible to manipulate controls extensively without the TCD driver ever knowing

[5] The terms "driver" and "command" are almost interchangeable. In this article, when I mean the whole mechanism that records and plays back the transaction, I use the term "driver." When referring more specifically to the details of how to use that driver in a script, I use the term "command."

Figure 8 Recording with the TCD and SAPGUI Drivers



that anything is going on, and this, of course, makes testing transactions that contain these controls almost impossible.

The answer to this problem, as **Figure 8** shows, was to incorporate a new recording technique *within* the SAP GUI. Instead of waiting for a whole batch of information to arrive on the application server, as the TCD driver does, the *SAPGUI driver* catches user actions as they happen at the frontend and passes them back to eCATT. In this way, eCATT has a record of what the user is doing in the GUI window, regardless of the way in which the application is implemented. This makes it possible to test transactions that contain controls, and also allows you — for the first time — to test list-based transactions, where you can drill down through one or more levels of detail to reach the screen that contains the information you need.

Since in our example we're recording a transaction that uses controls, we will choose the SAPGUI driver. When using this driver, there are a couple of administrative tasks you need to perform to configure the recording itself, so let's take a look at these first.

Configure the Recording

To select the driver for the recording, in the "Editor" tab of the eCATT test script screen choose the "Pattern" button in the application toolbar. This function is similar to the statement pattern function in the ABAP Editor, which lets you construct ABAP commands without having to type them in yourself. In the "Insert statement" dialog shown in **Figure 9**, select the driver you want to use — in our case, "SAPGUI (Record)" — from the command dropdown list.

The name of the command interface, which displays the recorded transaction data, is proposed automatically ("SAPGUI_1" in the example). You can overwrite the proposed name if you wish, but there is normally no reason not to accept it. You must then pick a target system to specify where the test will be running. You can use any target system that you have defined in your system data container. Here I selected the "R3ENTERPRISE" target system defined in Figure 5. Click ✔, which launches the dialog shown in **Figure 10**. Here you specify the configuration of the individual SAPGUI commands that will later be generated in the test script. You can

Figure 9 *Select the SAPGUI Driver for Recording*

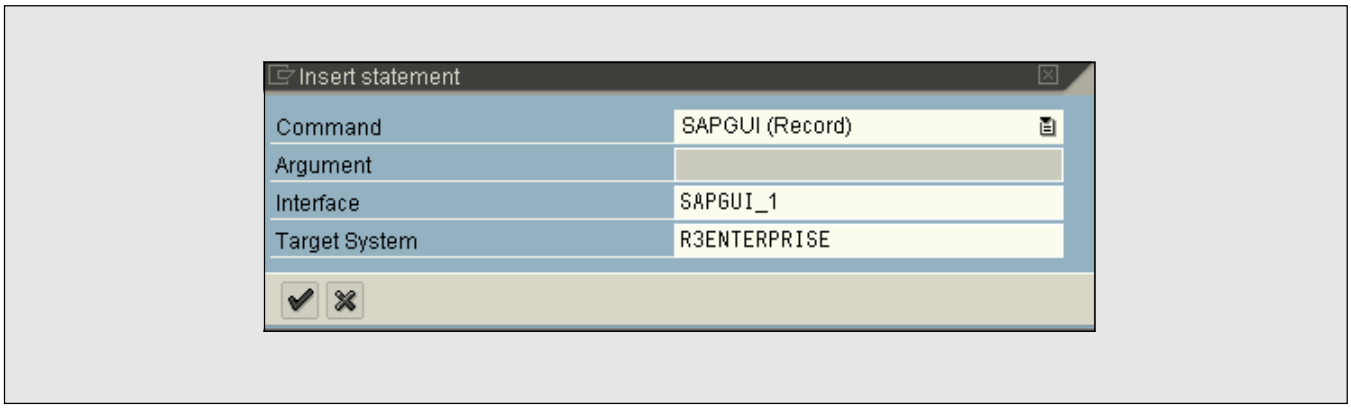
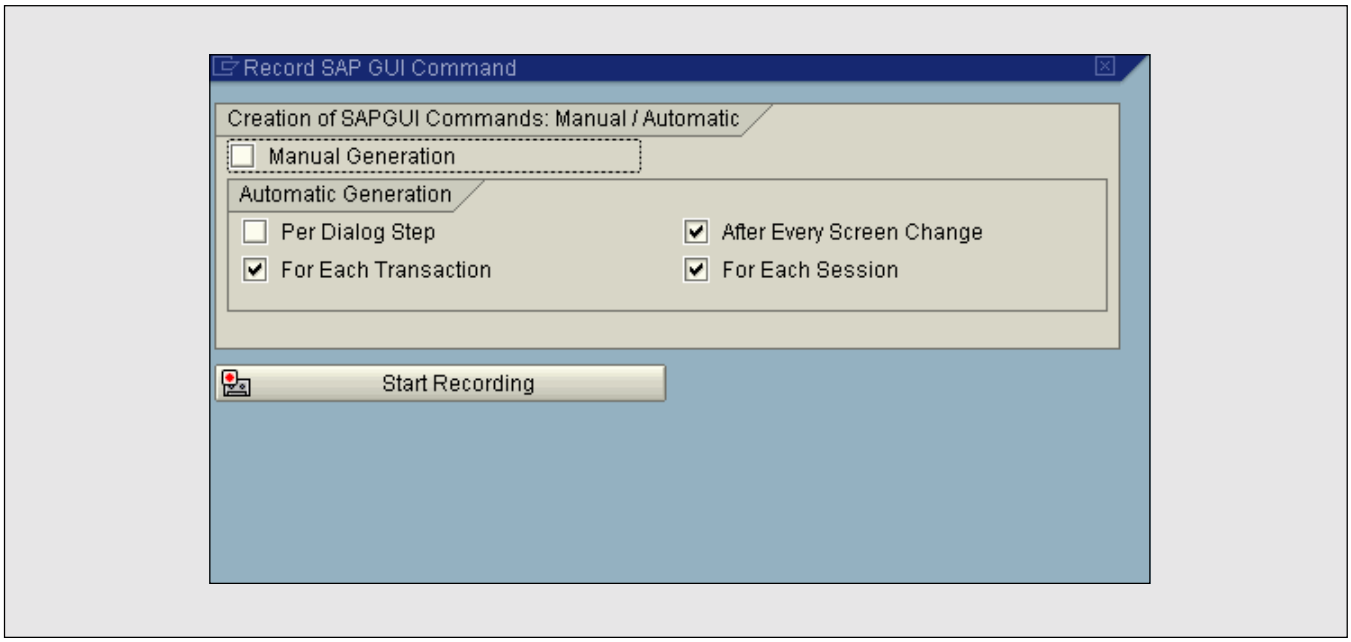


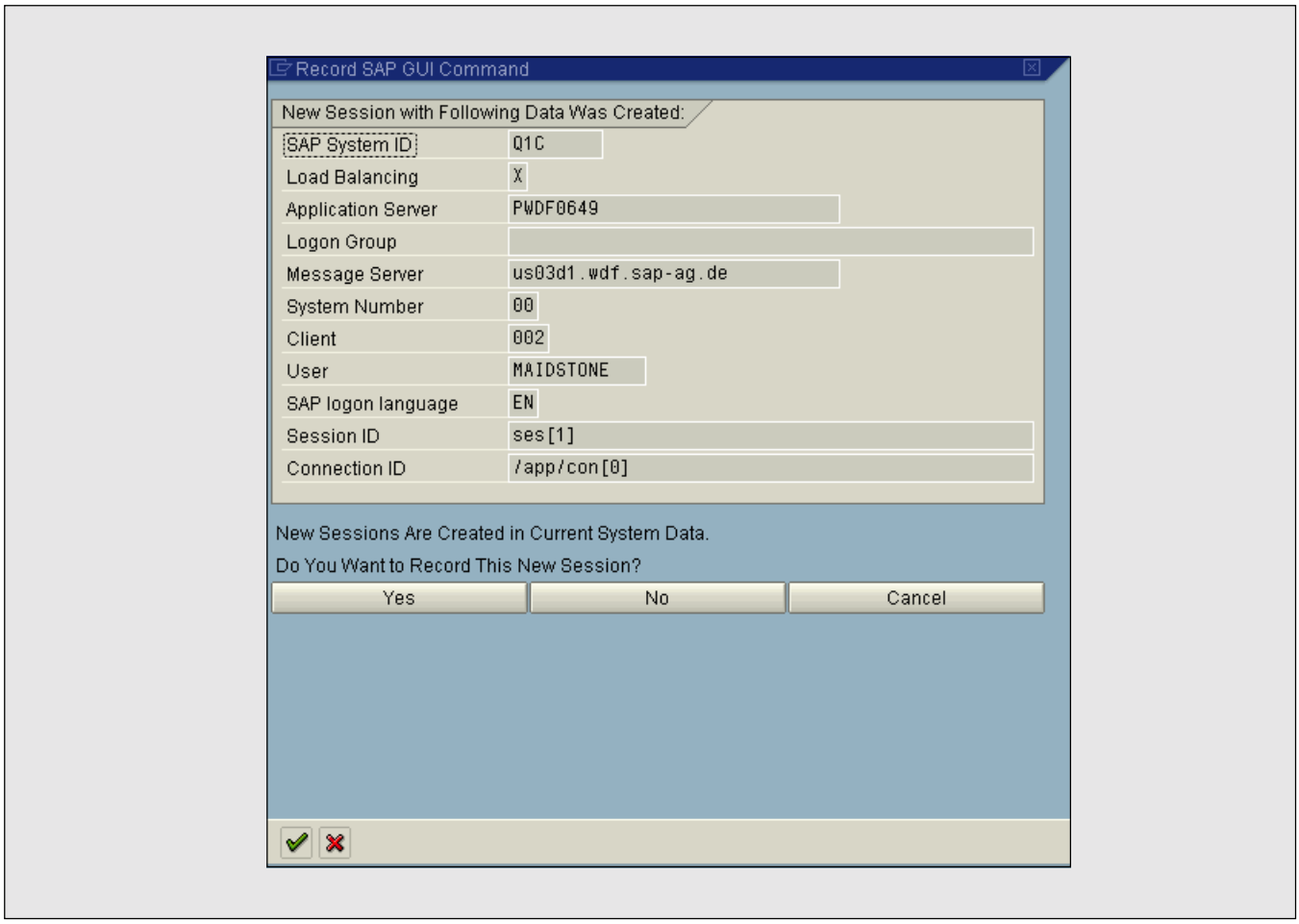
Figure 10 *Setting the SAPGUI Command Specifications*



choose to have a small number of commands, each of which contains a lot of information (for example, having a single SAPGUI command for each transaction, or even for a whole session), or you can choose to have a large number of commands, each of which contains relatively little information (for example, an SAPGUI command for each screen, or even for each user interaction).

The level of granularity that you choose is a matter of personal taste. However, keep in mind that in order to replay the recording with various combinations of field values for testing purposes, you will later have to edit the contents of the SAPGUI commands (i.e., the recorded changes) in the command interface. Consequently, I recommend that you use the first option, because the smaller the amount of data in the command interface, the less likely you are to put a value in the wrong place. The setting "After Every Screen Change" is generally a good choice. If you do not need to make any changes (because, for example, you will always run the transaction with the

Figure 11 Start a New Recording Session



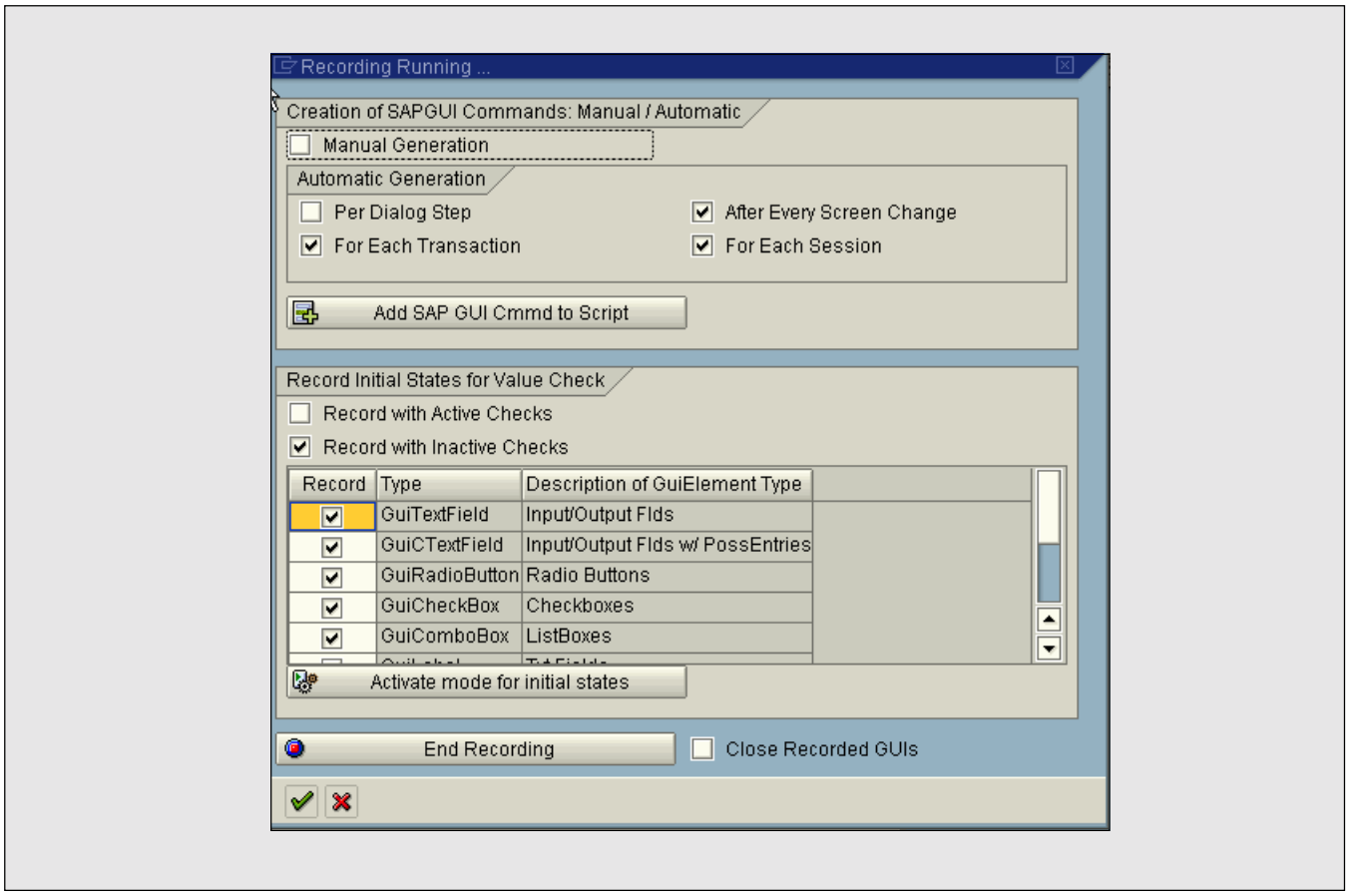
same set of values), then "For Each Transaction" can also be a good choice. Note that the settings are cumulative — if you select "After Every Screen Change," the two superordinate options "For Each Transaction" and "For Each Session" are automatically also selected.

> ✓ ***Tip***
>
> *Having a new SAPGUI command for each screen gives you a manageable amount of data to work with in each command interface.*

Once you have set these granularity levels, choose the "Start Recording" button. eCATT starts a new session on the chosen target system, and then displays the dialog box shown in **Figure 11** and asks if you want to record the new session. All of the fields are filled automatically by the system, so all you need to do is choose "Yes," which takes you to the screen in **Figure 12**.

By default, the SAPGUI command records only a user's *actions* within a transaction screen, and not the contents of any fields on the screen. This is fine for just navigating through an application (for example, to perform a simple task for setting up part of your test environment), but it does not help if you want to

Figure 12 Specify Desired Screen Elements for Recording



retrieve the values from any output fields set by the application, or if you want to check that field contents are set correctly. At this point in the recording procedure, you can opt to record the contents of various kinds of user interface elements, in particular input/output fields and field labels. This can significantly swell the size of the command interface, though, so use this recording option with care.

The types of GUI elements you can select for recording are listed in the table in the "Record Initial States for Value Check" frame in Figure 12. Above this table are two options: "Record with Active Checks" and "Record with Inactive Checks." These options relate to how the recorded transaction is played back. If you choose *active* checks, the contents of the selected fields (i.e., their "initial states") are checked each time you run the script. The default value against which they are checked is the value that you entered in that particular field when you recorded the script, but you can also assign variable values (for example, if you want to check that the value in a date field is always today's date). If the values differ, the script will fail. If you choose to record using *inactive* checks, on the other hand, the field contents are recorded, which allows you to access them from your script, but they are not checked. At this stage, you activate or deactivate checks for certain types of elements (e.g., input fields), which will apply to all instances of that type during the recording. However, when you edit the recorded data in the command interface (SAPGUI_1 here), you can switch checks on or off for individual screen elements (e.g., a *particular* input field) by setting the "Check" flag in the element attributes (more on this in an upcoming section).

It depends on the nature of your test as to whether you use active checks, inactive checks, or no checks at all. Sometimes you will not be able to check field values because you do not know precisely what value should occur in a field. At the same time, there is no sense in enabling a check on a field that is filled by an internal number assignment, since the value can never be the same in two consecutive script runs.

As you can see in Figure 12, I've selected "Record with Inactive Checks" and I've also opted to record the initial states of a range of user interface elements, including input/output fields. Having selected one of these two recording options, along with the screen elements you would like to record, you must choose "Activate mode for initial states" to communicate to the SAP GUI that it must record the field contents as well as the user's actions. (If you opted not to record any types of screen elements, you do not need to click this button. Instead, just switch to the new session that was created when you clicked the "Start Recording" button in Figure 10.)

So far you have seen how to configure the SAPGUI command to record a transaction and implement checks for field values. Time to perform the actual recording.

> ✓ ***Tip***
>
> *To minimize the amount of data included in the command interface, only record the initial states of the screen element types that you really need.*

Perform the Recording

With all of the necessary specifications in place, we are now ready to record the example flight-booking transaction. As with the TCD driver, work through the transaction as you normally would. As you go along, SAPGUI commands record your actions and, if you selected any screen elements in Figure 12, the contents of the selected field types in the application.

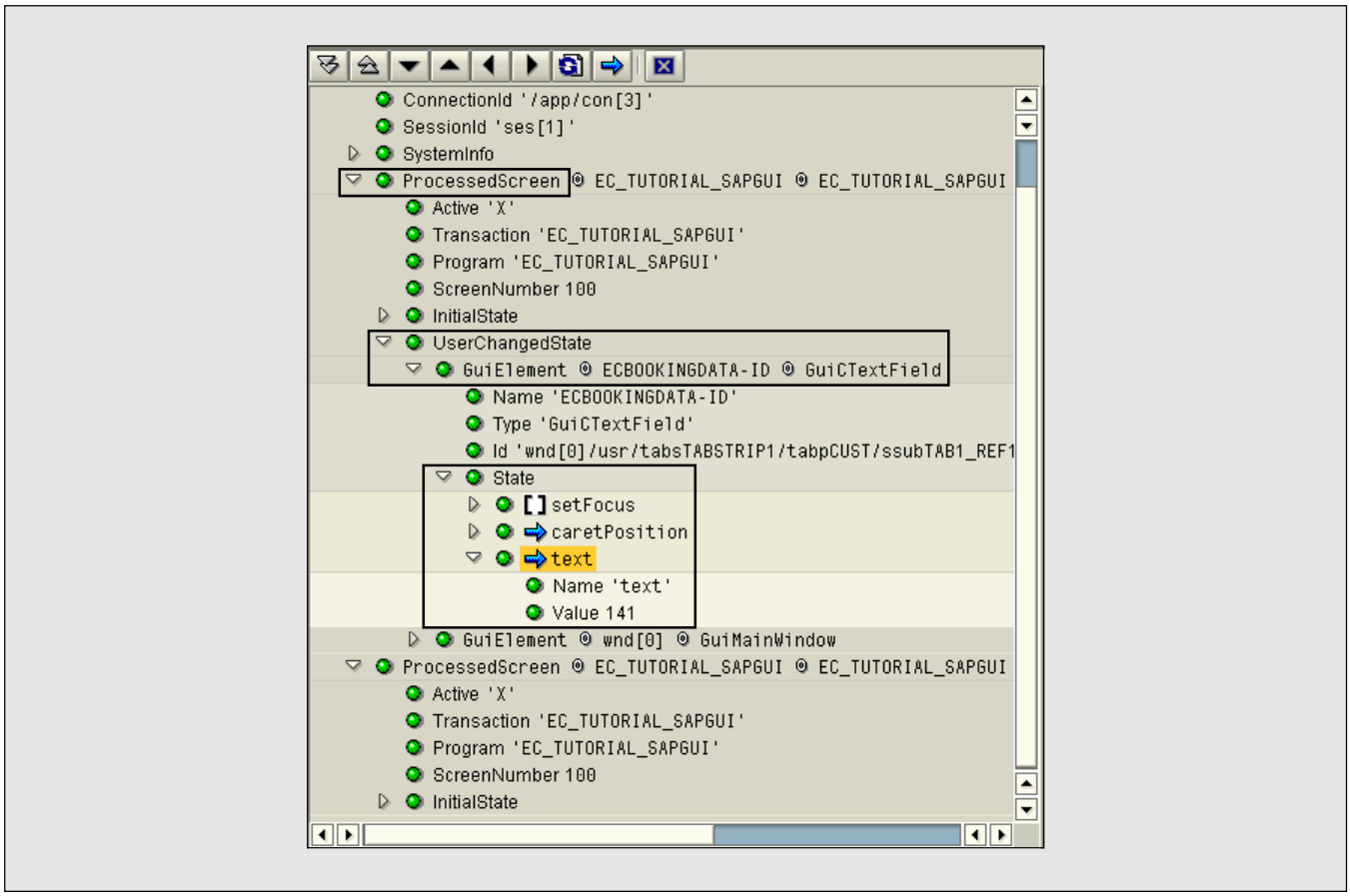
When you are finished working through the application, return to the eCATT dialog box (Figure 12) and click "End Recording." This stops the recording and returns control to eCATT, where you can set import and export parameters for testing.

Parameterize the Recorded Data for Testing

Once the recording has ended, go to the test script maintenance screen and select the "Editor" tab. You will see that the SAPGUI commands describing the recorded data have been inserted into the test script, and just as you can with the TCD command, you can further parameterize the data in order to replay the recording with various combinations of field values. To view or edit the actions recorded in an SAPGUI command, simply double-click the name of the command interface ("SAPGUI_1" in the example). eCATT opens up the command interface in a special structure editor on the right-hand side of the screen. In this section, I will show you the most important parts of the command interface and some of the things that you can do with the recorded data it displays.

When you look at an SAPGUI command interface, like the one in **Figure 13**, you will see that it is divided up into different nodes, each of which contains a number of subnodes. The most important part of the command interface is the "ProcessedScreen" node. There is one "ProcessedScreen" node for each screen in your application, so if you record a transaction that contains screens 100, 200, and 300, you would have three of these nodes. Within each "ProcessedScreen" node is a subnode called "UserChangedState," which lists all of the actions you performed during the recording. Figure 13 shows that I made changes in a field called "ECBOOKINGDATA-ID" (the field in the flight-booking application that contains the customer number) while recording the example transaction. Going further down the hierarchy to the "State" subnode, under "text" you can see the value I entered during the recording (customer number "141").

Figure 13 The SAPGUI Command Interface Display of Recorded Data



The next time I run the script, I might want to try forcing an error in the application by specifying a customer number that doesn't exist. For example, in Figure 13, you could parameterize the value of field "ECBOOKINGDATA-ID" by replacing the value "141" with a new string value or variable. Simply double-click the value and enter the new value or variable name in the "Value" field of the pop-up table that appears. The next time you run the script, eCATT will use the new value. If the variable is an importing parameter of the script, you can fill it with different values by creating test data containers for variant data and then linking the containers to the script via a test configuration, which you can configure to replay the script several times over.[6]

[6] For details on creating test data containers and test configurations, refer to SAP's online help for SAP Web Application Server 6.20.

However, as you look at the "UserChangedState" entries, you will quickly realize that these nodes provide no information about the contents of screen fields that you *did not change yourself*. Consequently, you have no way (in the "UserChangedState" section, at least) of retrieving the contents of output fields on screens. This could be fatal to your test activities, since output fields in transactions can contain vital information about whether an application is running as you expected (for example, if you look up a customer using a customer number, you might want to read the name, which the system displays in an output field).

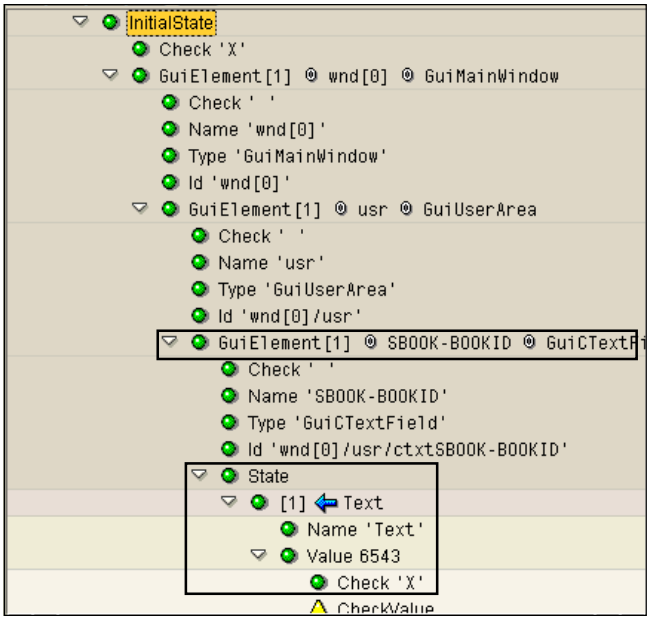
The contents of output fields that are set by the system are only recorded if you specifically requested them, like we did back in Figure 12. When you do this, eCATT generates an extra subnode in each

"ProcessedScreen" node called "InitialState" (see **Figure 14**). Here, you will find all of the field contents that eCATT recorded for you. Under each recorded element, you will see a subnode "State," which contains the attributes of the element in question. One of the attributes is "Text," which contains the field contents. Figure 14 shows the recorded contents of an output field "SBOOK-BOOKID" on one of the screens of the example transaction. Under this field's "State" subnode, you can see the various field attributes, the most important of which is "Text." In the "Value" subnode under "Text," you can see the actual value that was contained in the field ("6543" in the example).

To parameterize these values, you could double-click the "Text" subnode, which pops up a small table next to the tree structure with two rows: "Name" and "Value." If you enter a variable in the "Value" row, like "BOOKINGNUMBER," eCATT will place the value of the field into that variable when you run the script, enabling you to reuse the variable in later parts of the script. In the case of the booking number, let's say you want to run a test case where you first create a booking and then change it. After retrieving the booking number from this particular transaction (Create Booking), regardless of the driver you are using to record the transaction, you can reuse the variable to identify the booking by its number in the following transaction (Change Booking).

Beneath the "Value" subnode there are two further subnodes: "Check," which I mentioned earlier, and "CheckValue." "Check" specifies whether the field value should be checked at runtime (against the recorded value) and "CheckValue" is the value (i.e., part of the "initial state") against which the check is performed. The default value for the check is the value that was contained in the field when you recorded the transaction. However, you can parameterize the value if you need to. Note that even if "Check" is set to "X," the check will not be performed if a check field further up the hierarchy is set to " " (space), so if you find that your checks are not being executed, take a look at the "Check" flags higher up in the tree.

Figure 14 The "InitialState" Subnode of the SAPGUI Command Interface

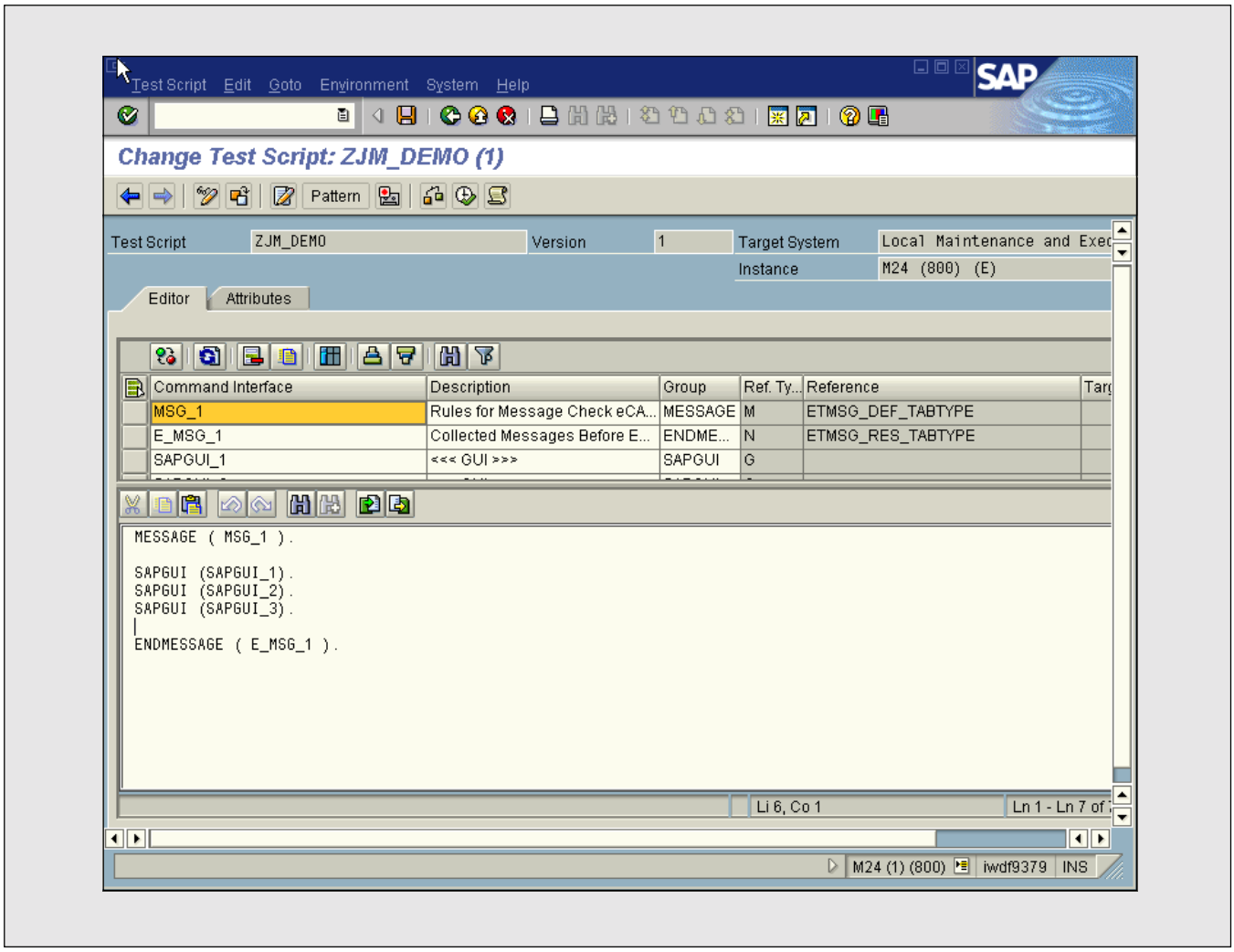


Define Message-Handling Rules for Testing

Messages are important within a transaction — not only do they notify us of errors, they also sometimes display information that is not available elsewhere (for example, a purchase order number in a message like "Order 100004 created"). When testing transactions, it is important to have a flexible message-handling mechanism. Sometimes we will require a particular success message as proof that the transaction is working. On other occasions, we may want to tolerate error messages (rather than allow them to cause the transaction to fail). We may even be expecting an error message as the *correct* outcome of a test case — suppose, for example, that the aim of the test is to ensure that a reasonable error message appears if demands on a particular resource (material, warehouse space, etc.) exceed the available supply.

For the SAPGUI command, there is a new message-handling construction that allows you to

Figure 15 The MESSAGE Command Interface



specify highly flexible rules for message processing.[7] A rule can apply to a particular type of message ("all error messages"), or it can apply to messages from a particular message class ("all messages from message class ZJM") or a particular message number ("message ZJM 002"). Each rule specifies whether the message or messages to which it applies are required, expected, allowed, or forbidden.

You specify the rules in the command interface of the MESSAGE command, which begins a MESSAGE … ENDMESSAGE processing block (see **Figure 15**). Any message that occurs within the processing block is tested against the rules that you specify and is handled according to your specifications. There is also an ENDMESSAGE command interface, which delivers a list of all of the messages that occurred within the processing block.

The MESSAGE command interface ("MSG_1" in the example) is a table. Each line of the table specifies a single rule.

[7] In the next eCATT release, this mechanism will be extended to include FUN, TCD, and ABAP blocks as well as the SAP GUI.

Figure 16 Message-Handling Rules

Field	Description
MODE	Specifies whether the message in question is: • Required (R) • Expected (E) • Allowed (A) • Forbidden (F)
EXIT	Specifies what should happen if the message occurs: • Empty: Continue processing within the MESSAGE … ENDMESSAGE block • "X": Jump to the corresponding ENDMESSAGE command and continue processing there
MSGTYP	Specifies the message type (S, I, W, E, A, X)
ID	Class of the message to which the rule applies
NUMBER	Number of the message to which the rule applies

The rules are summarized in **Figure 16**.

In addition to the rules that you define, there are also certain default rules — namely that messages with type S, I, or W are defined as successful, while types E, A, and X will cause the script to fail. However, any rules that you define take precedence over the default rules.

When a message occurs in the transaction, the system deals with it as follows:

1. The message is checked against the rules in the MESSAGE command interface in the order in which the rules are defined.

2. The message is checked against the default rules.

Figure 17 contains an example of two rule messages, each of which is set to mode "E" (expected). The particular messages to which the rules apply are message number "011" from message class "ZJM" and message number "012" from the same class. When the script is executed, each message is checked against the rules. In simple terms, eCATT asks, "Are you message 011 from message class ZJM?" If the answer is "No," eCATT asks, "Are you message 012 from message class ZJM?" The script will pass if one of the commands within the MESSAGE ... ENDMESSAGE block sends *either* message ZJM011 *or* message ZJM012. To require *both* ZJM011 *and* ZJM012, the mode would have to be set to "R" (required).

It is important to note that as soon as the system encounters a rule that applies to the current message, *no further messages are processed*. Consequently, the order in which you define your messages can be crucial.

Figure 18 contains an example of bad practice in this respect. Look at the first rule in the list. It is designed to *fail* the script when *any* message from message class ZJM occurs — even messages 011 and 012, which have different rules. Since the rules are processed in the order in which you define them, the second and third rules in this example can never be applied to the messages for which they are intended. If the real intent behind the rules is "*expect* that *either* ZJM011 *or* ZJM012 must occur, but *fail* the

Figure 17 *Example Rule Messages*

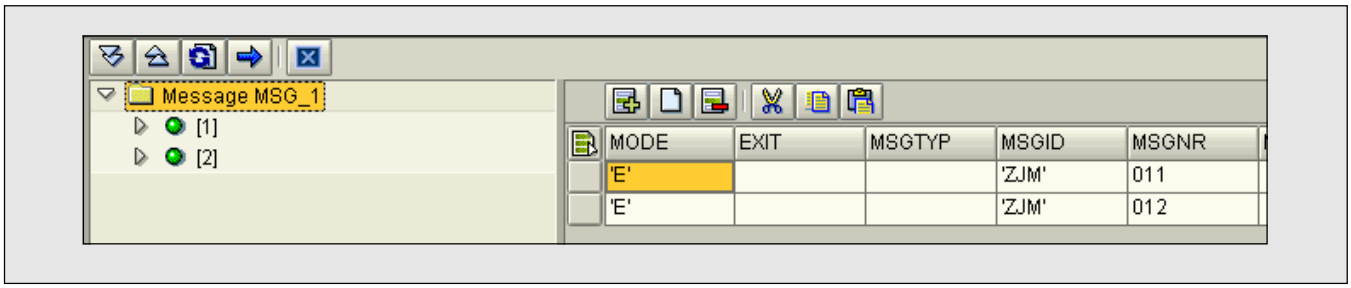


Figure 18 *Badly Defined Message Order*

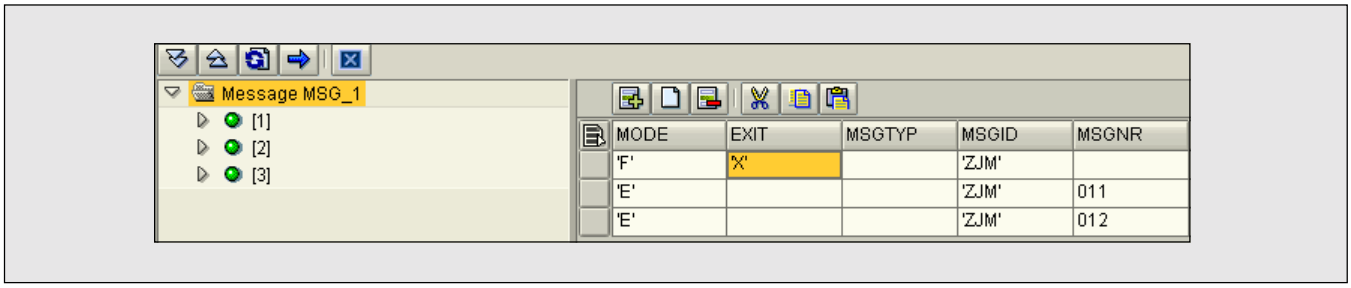
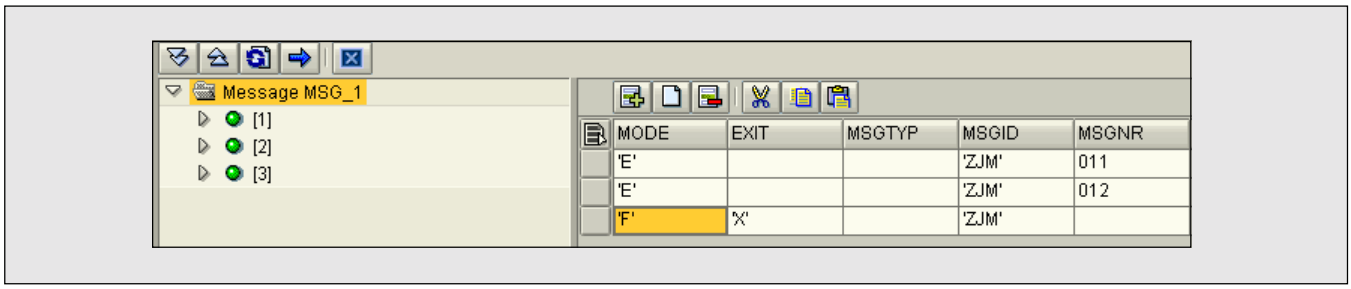


Figure 19 *Well-Defined Message Order*



script if *any other messages* from message class ZJM occur," then the most general rule ("fail the rest") must occur *after* the specific rules, as shown in **Figure 19**.

The ENDMESSAGE command interface contains a table of all the messages that occurred within the MESSAGE … ENDMESSAGE processing block. From this object, you can access information such as the variables from the message. This is important, since crucial information (such as customer number, order number, or other key data that you can use to identify a person, object, or resource) is often output in a message.

Step 4: Execute the Test Script

Once you have completed your test script, you can execute the test from the script editor. The result of each test run is a log, which shows you whether

Additional Information

- ☑ The *SAP Insider* article "eCATT — The New Test Tool for Your E-Business Solutions" in the July-September 2002 issue (available in the Article Archives at **www.SAPinsider.com**) provides an overview of eCATT.
- ☑ The online eCATT documentation at **http://help.sap.com** (choose *SAP Web Application Server 6.20 → mySAP Technology Components → SAP Web Application Server → Computer Aided Test Tool → eCATT: extended Computer Aided Test Tool*) provides detailed information on the tool, including full details on the eCATT scripting language, along with a listing of the new commands and examples.

the test script run has passed or failed. If the script run fails, it is usually due to one of the following conditions:

- A transaction sends an error message
- A check on a field value fails

In either of these cases, the log will document what happened, and you can go back and make any necessary changes to the message-handling rules you specified and the parameters you defined.

Another reason a script might fail is if the application itself has changed (for example, a developer has built new screens into the application that were not present when you created the recording). In this case, your script does not have enough information to "drive" the new screens — it does not know what fields there are on the new screens and, even if it did, it would not be able to guess the values that you might have put in them! In this case, you will actually have to re-record the application in order to make the script work again.

Once you've got your test script in good working order, you're ready to use it productively in your environment by creating test data containers for your variant test data, and a test configuration that links these test data containers to your test script. While these tasks are beyond the scope of this article, you can find more details in the online help for SAP Web Application Server 6.20 (see the sidebar to the left).

Conclusion

Test results validate that your system is customized correctly and that transactions and components are properly integrated, or give you fair warning that they are not. Using an automated test tool in addition to human testers, who can spot things that are not explicitly included in a test plan, can save you considerable time and effort.

By combining test scripts and sets of test data into test configurations, you can create executable data-driven test cases that can be incorporated in test catalogs and assigned to individual testers in the Test Workbench. The Test Workbench also includes monitoring tools that provide a real-time overview of each test project. eCATT thus provides you with a mixture of tried-and-trusted functions and cutting-edge technology that allows you to extend the range and reduce the costs of your SAP testing activities.

Jonathan Maidstone graduated from the University of Bristol (UK) in 1996 with a degree in modern languages and joined SAP as a translator for the ABAP Language and ABAP Workbench departments. Following a spell as a technical writer, during which he was involved with the SAP Control Framework and DCOM Connector projects, he is now a product management specialist with particular responsibility for rolling out eCATT, SAP's new test tool. Jon can be contacted at jonathan.maidstone@sap.com.

