Automated Test Code Generator (ATCG) Reference

VERSION 2018.0.01

Contents

Before You Begin

The *JADE Automated Test Code Generator (ATCG) Reference* is intended as a major source of information when you are using the ATCG to record and replay Graphical User Interface (GUI) actions in JADE applications.

Who Should Read this Reference

The main audience for the *JADE Automated Test Code Generator (ATCG) Reference* is expected to be developers or testers of JADE application software products.

What's Included in this Reference

The *JADE Automated Test Code Generator (ATCG) Reference* has four chapters.

Chapter 1	Gives an overview of the Automated Test Code Generator
Chapter 2	Getting started
Chapter 3	ATCG development structure and considerations
Chapter 4	Design guidelines and directions

Conventions

The *JADE Automated Test Code Generator Reference* uses consistent typographic conventions throughout.

Convention	Description
Arrow bullet (»)	Step-by-step procedures. You can complete procedural instructions by using either the mouse or the keyboard.
Bold	Items that must be typed exactly as shown. For example, if instructed to type **foreach**, type all the bold characters exactly as they are printed.
	File, class, primitive type, method, and property names, menu commands, and dialog controls are also shown in bold type, as well as literal values stored, tested for, and sent by JADE instructions.
Italic	Parameter values or placeholders for information that must be provided; for example, if instructed to enter *class-name*, type the actual name of the class instead of the word or words shown in italic type.
	Italic type also signals a new term. An explanation accompanies the italicized type.
	Document titles and status and error messages are also shown in italic type.
Blue text	Enables you to click anywhere on the cross-reference text (the cursor symbol changes from an open hand to a hand with the index finger extended) to take you straight to that topic. For example, click on the "Persistent Data State" cross-reference to display that topic.
Bracket symbols ([ ])	Indicate optional items.
Vertical bar (|)	Separates alternative items.

Convention	Description
`Monospaced font`	Syntax, code examples, and error and status message text.
ALL CAPITALS	Directory names, commands, and acronyms.
SMALL CAPITALS	Keyboard keys.

Key combinations and key sequences appear as follows.

Convention	Description
KEY1+KEY2	Press and hold down the first key and then press the second key. For example, "press Shift+F2" means to press and hold down the Shift key and press the F2 key. Then release both keys.
KEY1,KEY2	Press and release the first key, then press and release the second key. For example, "press Alt+F,X" means to hold down the Alt key, press the F key, and then release both keys before pressing and releasing the X key.

Related Documentation

Other documents that are referred to in this reference, or that may be helpful, are listed in the following table, with an indication of the JADE operation or tasks to which they relate.

Title	Related to…
JADE Developer's Reference	Developing or maintaining JADE applications
JADE .NET Developer's Reference	Developing applications using .NET class libraries exposed in JADE
JADE Database Administration Guide	Administering JADE databases
JADE Development Environment Administration Guide	Administering JADE development environments
JADE Development Environment User's Guide	Using the JADE development environment
JADE Encyclopaedia of Classes	System classes (Volumes 1 and 2), Window classes (Volume 3)
JADE Encyclopaedia of Primitive Types	Primitive types and global constants
JADE Installation and Configuration Guide	Installing and configuring JADE
JADE Initialization File Reference	Maintaining JADE initialization file parameter values
JADE Object Manager Guide	JADE Object Manager administration
JADE Report Writer User's Guide	Using the JADE Report Writer to develop and run reports
JADE Synchronized Database Service (SDS) Administration Guide	Administering JADE Synchronized Database Services (SDS), including Relational Population Services (RPS)
JADE Thin Client Guide	Administering JADE thin client environments
JADE Web Application Guide	Implementing, monitoring, and configuring Web applications

Chapter 1 Automated Test Code Generator (ATCG)

This chapter covers the following topics.

Overview

The Automated Test Code Generator (ATCG) enables you to record and replay GUI actions in JADE applications, by capturing the execution of GUI event methods and generating code to replay those actions.

JADE's method tracking enables you to identify a target method to be tracked and to intercept the execution of that method. (For details about method tracking, see Chapter 18, "Tracking Methods", of the *JADE Developer's Reference*.)

At recording time, code snippets are generated. When the recording session is finished, a **.cls** file is generated and loaded into the database. When run with the standard functionality (that is, using the **Replay Last** or **Replay List** button on the ATCG Control dialog), the original GUI actions are replayed.

ATCG is typically used to create regression tests or benchmark systems. Although it has many uses, its primary purpose is to generate code that you can use in data-driven testing.

Although there are a number of tools on the market that claim to allow the recording and replay of GUI actions for regression testing, they require the use of an additional language, and they are frequently complex and expensive. In addition, it can be difficult to specify in these tools which differences in the tested application are superficial and which are important.

The output from ATCG is JADE code, which you can modify to meet your requirements. For example, if a button on a form is renamed between releases of a regression test, you can easily find and fix the affected code. When used to create a benchmark system, you can record a business transaction once, and by modifying the generated JADE code, replay it many times in a loop; for example, reading input data from a flat file, or looping through a table and handling all rows.

Structure

ATCG requires the **AtcgGeneratorSchema** schema and the **AtcgTestCodeSchema** schema.

Load the:

1. **AtcgGeneratorSchema** under the schema in which the tested application runs, because the recording and replay both must execute in the context of the original application
2. **AtcgTestCodeSchema** schema under the **AtcgGeneratorSchema** schema

Tip You can install multiple pairs of **AtcgGeneratorSchema** and **AtcgTestCodeSchema** schemas in the same JADE database, as long as each pair has different names.

Applications

The applications that form ATCG are listed in the following table.

Application	This application...
AtcgRecordApp	Is displayed like the main form of the application that is being tested, and is under your control. GUI actions are recorded in a log file by this application. This is essentially the tested application, but it is run from the context of the **AtcgTestCodeSchema** schema rather than the context of the schema in which the application is defined.
AtcgReplayApp	Is displayed like the main form of the application that is being tested but it is under the control of the replay infrastructure. GUI actions are replayed by executing the generated JADE code.
AtcgControlApp	Controls the whole record and replay process by: 1. Enabling you to begin recording 2. Deciding which methods to track 3. Generating and loading the recorded code snippets as a profile class 4. Handling simple replay scenarios
AtcgBtnClick	Controls the buttons to click when replaying recorded actions to Windows common dialogs, by calling Windows Application Programming Interfaces (APIs).

ATCG Variants

The variants of ATCG are:

- Generic

  As the generic version of ATCG contains no code specific to any application, it requires installation actions before you can record test profiles.

- Erewhon Sample

  You can load and use this variant with the standard Erewhon schemas available from the **JADE-Erewhon** link at https://github.com/jadesoftwarenz.

Chapter 2 Getting Started

This chapter covers the following topics.

Downloading the ATCG Schemas

Download the ATCG schemas and associated forms definition files from the **JADE-ATCG** link at https://github.com/jadesoftwarenz.

Installing the Erewhon ATCG Variant

Before you install the generic version of ATCG, install the Erewhon variant so that you can refer to the provided examples.

To install the Erewhon variant

1. From the **JADE-Erewhon** link at https://github.com/jadesoftwarenz, load the Erewhon example schemas into JADE and familiarize yourself with the applications.
2. Enable method tracking, by updating your JADE initialization file so that the value of the **MethodTrackingEnabled** parameter in the [JadeSecurity] section is set to **true**.
3. Load the **AtcgGeneratorSchema** (that is, the **AtcgGeneratorSchema_Generic.scm** and **AtcgGeneratorSchema_Generic.ddb** or **AtcgGeneratorSchema_Generic.ddx** files) as a subschema of

ErewhonInvestmentsViewSchema, as follows.

a. Load the schema and forms definition files in the usual way, by specifying the schema and forms file names on the **File Selection** sheet of the Load Options dialog, and then clicking the **Advanced** button.

b. On the Advanced Load Options dialog:

   i. Change the **Subschema of** value from **RootSchema** to **ErewhonInvestmentsViewSchema**.

   ii. In the text box portion of the **Target Schema** combo box, specify **AtcgGeneratorSchema**. (This schema is *not* displayed in the drop-down list of the combo box.)

   iii. Click the **OK** button.

4. Load the **AtcgTestCodeSchema** (that is, the **AtcgTestCodeSchema_Generic.scm** and **AtcgTestCodeSchema_Generic.ddb** or **AtcgGeneratorSchema_Generic.ddx** files), by performing the actions specified in the previous step of this instruction, but specifying **AtcgGeneratorSchema** in step b.i and **AtcgTestCodeSchema** in step b.ii.

5. In **AtcgTestCodeSchema**, change the name of the **XxxProfile** class, a subclass of **AtcgProfile**, to **EDProfile** (that is, it is the class for the Erewhon Demonstration profile).

Note This **EDProfile** class will be the superclass for all of your profile test classes.

6. Modify the **Application** class **atcgGetControlOptions** and **atcgRecordAppInit** methods (in **AtcgTestCodeSchema**) with the new class name, by replacing instances of **XxxProfile** with **EDProfile**.

7. In the **AtcgTestCodeSchemaApp** class **atcgGetControlOptions** method, change the method so that the **ErewhonInvestmentsViewSchema** is tracked; for example:

```
....
// list of schemas to be tracked during recording
targetSchemas.add("AtcgTestCodeSchema");
targetSchemas.add("ErewhonInvestmentsViewSchema");
....
```

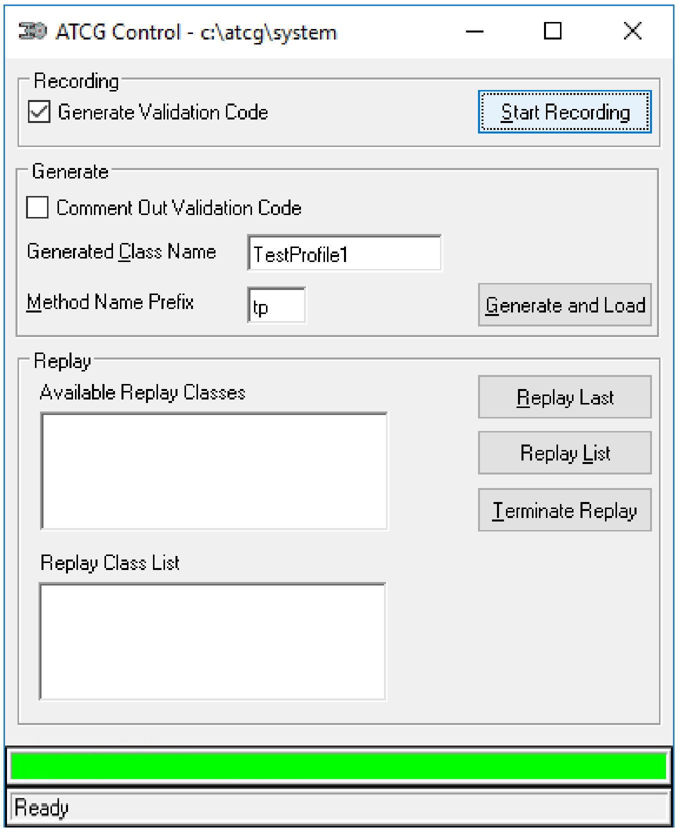
8. Modify the global **GErewhonInvestmentsViewSchema** class **getAndValidateUser** method in the **ErewhonInvestmentsViewSchema** so that all of the ATCG applications can run; for example:

```
if app.name <> AdminApp and
        app.name <> "AtcgControlApp" and
        app.name <> "AtcgReplayApp" and
        app.name <> "AtcgRecordApp" and
        ....
    return false;
endif;
....
if app.isWebShopApp or
    app.name = "AtcgControlApp" or
    app.name = "AtcgReplayApp" or
    app.name = "AtcgRecordApp" or
    ....
then
    return true;
endif;
```

9. Modify the **GErewhonInvestmentsViewSchema** class **isUserValid** method in the **ErewhonInvestmentsViewSchema** so that all of the ATCG applications can run; for example:

```
if app.isWebShopApp or
    app.name = "AtcgControlApp" or
    app.name = "AtcgReplayApp" or
    app.name = "AtcgRecordApp" or
    ....
then
    isValid := true;
endif;
```

10. Check that the **AtcgControlApp** application in **AtcgTestCodeSchema** runs; that is, the dialog like the example in the following image is displayed.



Generated Class Name: TestProfile1

Method Name Prefix: tp

Ready

11. To get the **AtcgRecordApp** working, modify the **startup** method in the **EDProfile** class (a subclass of **AtcgProfile** in **AtcgTestCodeSchema**) to create the main form and log-on screen for the **ErewhonShop** application.

```
vars
    c : Client;
    mf : FormShopSaleItems;
begin
    app.atcgLogMessageTC("===========");
    app.atcgLogMessageTC("Starting profile");
    app.atcgLogMessageTC("===========");

    // create and show main form and log-on screen
    create mf;

    // This is just a quick example by hard-coding to the first client.
```

```
        c := Client.firstInstance;
        app.setClient(c);
        mf.show;
        app.atcgLogMessageTC(method.qualifiedName&" finished");
    end;
```

12. Start the **AtcgRecordApp** application manually and then:

    a. Check that the **ErewhonShop** application in the **ErewhonInvestmentsViewSchema** can be started.

    b. Close the application.

    Note **AtcgRecordApp** must be run as a standard (fat) client.

13. Load the provided **BasicDemo.cls** example profile class into **AtcgTestCodeSchema**.

14. Start the **AtcgControlApp** application.

    a. Double-click on **Basic Demo**.

    b. Click on the **Replay List** button, which should replay a basic sequence of events in the **ErewhonShop** application.

    If the setup is correct and the data matches the recorded profile, this recorded sequence should replay correctly.

15. Click the **Start Recording** button, to start the **AtcgRecordApp** application automatically. When the Please Wait form closes:

    a. Perform a couple of actions within the application.

    b. Once completed, click the **Generate and Load** button to terminate the **AtcgRecordApp** application and generate and load a new profile class.

    c. Click the **Replay Last** button. The main form of your **ErewhonShop** application should be displayed and the actions that you performed then replayed.

Basic installation is now complete.

Note Further modifications may be required to automatically handle items such as modal forms. For details, see Chapter 4, "Design Guidelines and Directions".

Installing the Generic Version of ATCG

Before you install the generic version of ATCG, install the Erewhon variant so that you can refer to the provided examples (for details, see "Installing the Erewhon ATCG Variant", in the previous section).

You can have multiple instances of the Automated Test Code Generator in your JADE system, as long as each instance has a different name.

Caution If you do not specify a unique profile and schema name for each schema in the JADE system into which you load ATCG, when you record and replay GUI actions using the control application (that is, the **AtcgTestCodeSchemaApp** class **atcgControlAppInit** method), ATCG freezes up and repeatedly generates unhandled exceptions.

If you load more than one instance, rename each one within the **Application** class **atcgGetControlOptions** method in each schema into which you load it; for example:

```
/* the schema and its superschema into which to generate profile classes; for
example, DSProfile and DSAtcgGeneratorSchema when targeting DummySchema */
genSchema := "DSAtcgTestCodeSchema";
genSuperSchema := "DSAtcgGeneratorSchema";
```

To install the generic version of ATCG

1. Back up your database.
2. Enable method tracking, by updating your JADE initialization file so that the value of the **MethodTrackingEnabled** parameter in the [JadeSecurity] section is set to **true**.
3. Load the **AtcgGeneratorSchema** (that is, the **AtcgGeneratorSchema_Generic.scm** and **AtcgGeneratorSchema_Generic.ddb** or **AtcgGeneratorSchema_Generic.ddx** files) as a subschema of the schema in which the target application runs.
4. Load the **AtcgTestCodeSchema** (that is, the **AtcgTestCodeSchema_Generic.scm** and **AtcgTestCodeSchema_Generic.ddb** or **AtcgGeneratorSchema_Generic.ddx** files) into the **AtcgGeneratorSchema** target schema.
5. In **AtcgTestCodeSchema**, change the name of the **XxxProfile** class, a subclass of **AtcgProfile**, to a prefix suitable for your application. This will be the superclass for all of your profile test classes.
6. Modify the **Application** class **atcgGetControlOptions** and **atcgRecordAppInit** methods (in **AtcgTestCodeSchema**) with the new class name, by replacing instances of **XxxProfile** with the profile class name that you selected in the previous step of this instruction.
7. In the **AtcgTestCodeSchemaApp** class **atcgGetControlOptions** method, enter the list of target schemas to be tracked; for example:

   ```
   ....
   // list of schemas to be tracked during recording
   targetSchemas.add("SecondDocExampleSchema");
   targetSchemas.add("TrainingExampleSchema");
   ....
   ```

   This array should include all schemas containing forms that are used by your application, including inherited and peer schemas. If there is one schema only, replace **XxxSchema** with the name of that schema.

   Note Many applications contain additional schemas such as **CardSchema** and **BaseSchema**, which should not be listed unless your application uses form classes defined in those schemas.

8. In the **AtcgTestCodeSchemaApp** class **atcgGetControlOptions** method, there is a list of "noise" methods to ignore. These are normally application methods that are called many times, and are not relevant to ATCG; for example, **app.convertStringToYesNo**. You may not know what to list until you have used ATCG for a while.

   Tip Do not list any control or form event methods, as that may affect code generation.

9. Check that the **AtcgControlApp** application in **AtcgTestCodeSchema** runs.

   This application should run without modification, unless there is something in your security code that prevents it; for example, your **global.getAndValidateUser** method may validate schema names or application names. While you are examining your security code, keep in mind that the **AtcgRecordApp** and **AtcgReplayApp** applications will also need to run in **AtcgTestCodeSchema**.

10. Get the **AtcgRecordApp** application working. This needs to run your target application, but it needs to run from within **AtcgTestCodeSchema** with the application name **AtcgRecordApp**, as follows.

    a. Reimplement the **app.initialize** method (which is called from the **app.atcgRecordAppInit** method) with code similar to your application's initialize method.

    b. Modify the method called **startup** in your profile superclass that you changed in step 3 of this instruction. This should create the main form and log-on form of your application.

    c. Change the method called **stop**, to log off and close the main form.

    For examples of the start and stop methods, see the sample Erewhon test code schema.

    When you can bring up the main form of your application and are logged in automatically, you can use the application as normal and close it down again cleanly, you are finished with this step.

11. Get the **AtcgReplayApp** application working, as follows.

    a. You will first need a sample profile class to run. Run the **AtcgControlApp** application, which needs to be run as a standard (fat) client.

       Note At this stage of installation, you should run the JADE development environment as a standard client and run all of the ATCG applications from there. Alternatively, you could create the appropriate shortcuts.

    b. In the **AtcgControlApp** application, click the **Start Recording** button, which starts the **AtcgRecordApp** application automatically.

    c. When the Please Wait form is no longer displayed, click the **Generate and Load** button. The **AtcgRecordApp** application will then terminate and a new profile class will be generated and loaded.

    d. Click the **Replay Last** button. The main form of your application should be displayed briefly.

Basic installation is now complete.

Note Further modifications may be required to automatically handle items such as message boxes and modal forms. For details, see Chapter 4, "Design Guidelines and Directions".

Using ATCG at Run Time

To be able to record properly, the **AtcgRecordApp** application must be run as a standard (fat) client. A presentation (thin) client cannot be used because many of the required GUI event method calls are suppressed if there is no explicit user code in them.

Single-user mode cannot be used because the load of the generated class file is done with an external batch schema load (**jadloadb**) standard client.

The **AtcgReplayApp** and **AtcgControlApp** applications can be run as standard (fat) or presentation (thin) clients. As the **AtcgControlApp** application starts the **AtcgRecordApp** application in its own node when using clean-start mode, the **AtcgControlApp** application can be used only be used in thin client mode only when you set the value of the **CleanStartMode** parameter in the [ATCG] section of the JADE initialization file to **false**. (This parameter is **true**, by default.)

Typically, the JADE development environment is run as a fat client, the **AtcgControlApp** application is started from the development environment, and the **AtcgRecordApp** and **AtcgReplayApp** applications are started from the **AtcgControlApp** application. All applications are therefore run in the same fat client node. All three applications run in the **AtcgTestCodeSchema**.

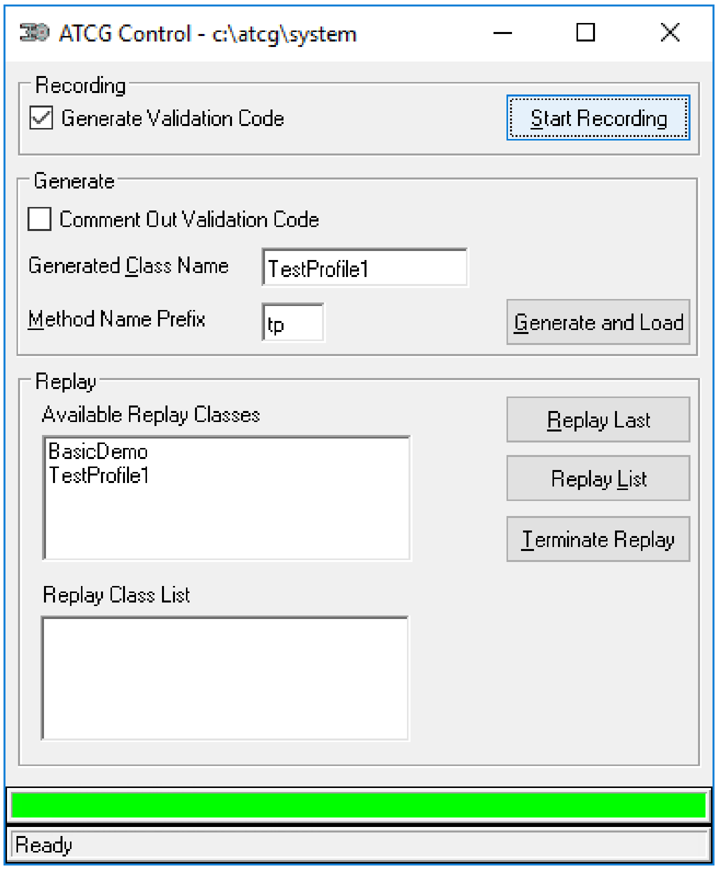
ATCG can be used with older releases of JADE, including JADE 2016 and JADE 7.1.

Example of Typical ATCG Usage

When you have installed and configured ATCG, you can record and replay GUI actions.

1. Initiate the **AtcgControlApp** application.

   The form similar to that shown in the following image is then displayed.


ATCG Control - c:\atcg\system

Recording
[✓] Generate Validation Code

Generate
[ ] Comment Out Validation Code
Generated Class Name: TestProfile1
Method Name Prefix: tp

Replay
Available Replay Classes:
BasicDemo
TestProfile1

Replay Class List

Ready

2. Click the **Start Recording** button, to start the **AtcgRecordApp** application.
3. Use the **AtcgRecordApp** application, which looks like the tested application normally does, to perform some actions; for example, using a navigation list box to open forms, double-clicking on table rows to display maintenance forms, using menu items, and other actions that you would normally perform in the application.
4. Click the **Generate and Load** button. (You can change the generated class name first, if required.)
5. When the load is complete, click **Replay Last**.

The tested application is started in the context of the **AtcgReplayApp** application, and it replays the actions just recorded in the **AtcgRecordApp** application. When the replay is finished, the application terminates.

Usage Considerations

This section covers the following topics.

Persistent Data State

Because the same actions are being repeated in the record and replay phases, issues can arise because of the state of the database. For example, if you added something with a unique identifier, the replay phase can fail with a duplicates exception.

You could back out the database changes manually, using your application after each record or replay session. For example, you could bring up a "vanilla" copy of your application and modify the data so that the replay action will work, or you could record your GUI actions again.

For simple testing scenarios, you may be able to simply record the adding of the unique item and deleting it all in one profile class. With this technique, you can record and replay any number of times without getting duplicates exceptions.

Running Nodes and Other Applications

There is generally no problem running other applications while recording and replaying GUI actions; for example, the tested application running in its normal schema context the usual way, the JADE Monitor, or the JADE development environment.

Note There can be only one **AtcgRecordApp** application running at a time.

Profile Class Size

It may be advisable to break up large amounts of work into multiple profile classes. That way, if you make a mistake, there is less that you have to re-record.

In addition, if you need to re-record parts of tests because of changes in the application, it is easier to do this if the profile classes are smaller.

Result of Unnecessary Record Actions

Some "nervous twitch" actions (for example, resizing or moving forms or unnecessary clicks) can cause additional code to be generated, which can make the generated profile classes more difficult to work with. It is best to avoid unnecessary actions while recording.

If the tested application makes a lot of calls to worker methods such as **app.showResponseAsYesOrNo**, these will not increase the size of the generated code but they may cause a lot of extra logging while recording.

You can suppress this logging, by adding these methods to the **noiseMethods** array in the **atcgGetControlOptions** method of the **Application** subclass. Some worker methods can be called many times from a single button click.

Ex_Abort_Action not Replayable

Replay of an exception handler returning **Ex_Abort_Action** is not supported. If this occurs during recording, a message box is displayed, stating that the replay action will stop when it gets to this point in the recorded profile.

Executing the AtcgReplayApp Application from the Command Line

To facilitate calling the **AtcgReplayApp** application from sources other than the **AtcgControlApp** application (for example, from a batch file or benchmark framework), you can specify the test class from the **jadclient** command line.

In the command line or shortcut to the application, specify the test class; for example, the following command line executes the **LogTransfers** test.

```
c:\atcg\bin\jadclient path=c:\atcg\system ini=c:\atcg\system\jade.ini schema=Atcg
app=AtcgReplayApp endJade ;LogTransfers;
```

Note The exit code is set to **4** for successful test execution and **1** for a test failure. An exit code of zero (**0**) can sometimes occur if the exit or terminate condition is unhandled.

Pausing on an AtcgReplayApp Error

The **AtcgReplayApp** application pauses the playback list if the pausing on error functionality is enabled and an error occurs.

To enable pausing on error, specify the following parameter with a value of **true** in the [ATCG] section of the JADE initialization file. (The default value is **false**.)

```
[ATCG]
PauseOnError=true
```

Note The **PauseOnError** parameter is ignored if the **AtcgReplayApp** application is executed from a source other than the **AtcgControlApp** application.

Validation Warnings for Date and Time Fields

The **AtcgReplayApp** application can return a warning instead of an error if a control contains a date or time value; for example, if a label caption contains **"Status as at 12:23"**, where **12:23** is the current time, validation will always fail.

The string is considered as possibly containing a date or time if there is a number followed by a slash (**/**) or colon (**:**) character followed by another number and there are no other intervening characters other than spaces. As this must be true for the expected string and for the actual string, the following examples are candidates for warnings rather than errors.

```
"Status as at 12:23"

"1/ 3/2009"

"Filed on 3/12"
```

The following examples are *not* candidates for warnings rather than error.

```
"01 Dec 2009"

"abc"

"1h23m14s"

"1.3.2009"
```

Enable the return of a warning instead of an error if a control contains a date or time value by specifying the following parameter with a value of **true** in the [ATCG] section of the JADE initialization file. The default value is **true**, but setting the value of the **ValNoWarningsOverride** parameter to **true** disables the date and time fields validation warnings.

```
[ATCG]
ValDateTimeWarningOnly=true
```

Note The **ValDateTimeWarningOnly** parameter is ignored if the **ValNoWarningsOverride** parameter is not set to **false**.

Validation Warnings for Text Boxes Associated with Tables

The **AtcgReplayApp** application can return a warning instead of an error if a control is a text box that has the value of the **automaticCellControl** property set to **true**.

When a **TextBox** control has the **automaticCellControl** property set to **true**, the **Table** control with which it is associated can update the text in the text box. This is done by the presentation client, and it is not always under the control of ATCG. If the associated table *does* update the text in a text box, validation may fail.

Enable the return of a validation warning instead of an error if the **automaticCellControl** property of a **TextBox** control is set to **true**, by specifying the following parameter with a value of **true** in the [ATCG] section of the JADE initialization file. The default value is **true**, but setting the value of the **ValNoWarningsOverride** parameter to **true** disables the warnings for text boxes associated with tables.

```
[ATCG]
ValAutoControlWarningOnly=true
```

Note The **ValAutoControlWarningOnly** parameter is ignored if the **ValNoWarningsOverride** parameter is not set to **false**.

Overriding Validation Warnings

The **AtcgReplayApp** application validation returns errors only, by default (that is, the default value of **true** disables the validation warnings for date and time fields and for text boxes associated with tables).

Disable the overriding of validation warnings so that warning messages *and* errors are returned by specifying the following parameter with a value of **false** in the [ATCG] section of the JADE initialization file.

```
[ATCG]
ValNoWarningsOverride=false
```

Chapter 3 Development Structure and Considerations

This chapter covers the following topics.

Overall Control

Most overall control is actioned from the ATCG Control form (the **AtcgControlForm** Form subclass in the **AtcgGeneratorSchema** schema).

The key methods in the **AtcgControlForm** class that are called from the button **click** event methods are listed in the following table.

Method	Description
doStart	Starts a recording session. It checks whether a session is already in progress, starts method tracking, notifies the record application, and so on.
doGen	Generates and loads the profile class. It stops the recording session, validates the specified class name, generates the **.cls** file from the **rrmsg.log** file, and loads it into the database.
doReplay	Validates the profile class to be replayed and handles replay lists.

Recording Actions

While recording, the **AtcgRecordApp** application writes code fragments and diagnostic information to **rrmsg<*nn*>.log**. This is a versioned JADE log file, which is rolled over each time it starts recording. All historical **rrmsg** logs are archived in the **logs** folder in case they are needed later; for example, for problem resolution.

When recording is started, form references are generated for all loaded forms, in case you decide to use them. Other form references are generated during capture of the preamble of the form's **load** method. In addition to the form reference itself, an application **getForm** method call is generated, to retrieve the form object identifier (oid) at replay time.

Note Most of the information written to the **rrmsg** log is written by the **AtcgControl** class **capture** method when operating under the tested application and method tracking.

Method Tracking

When a method such as the **load** method of an **AppMainForm** class is registered for tracking, JADE does a callback to the **capture** method just before and just after the execution of the **AppMainForm** class **load** method.

The **capture** method is therefore executing within the context of the tested application and it is effectively executed by the tested application but without it having to be changed to make the call. This means that:

- The **app** system variable is the same one referenced by the application code
- All transients are accessible
- The form and other objects are exactly as the application sees them when the tracked method executes

If the **Table** control **dblClick** is tracked, therefore, the **row**, **column**, **selected**, **text**, **itemObject**, and other attribute properties can all be accessed.

Logging Information

Some things are routinely logged, regardless of whether we plan to generate any code for them.

The following things are logged when entering a method (that is, the **capture** method is called as a preamble).

- Method name
- Class and oid of the receiver
- Name of the class in which the method is defined (it could be a superclass of the receiver)
- Any parameters of the method, including their type and value

The following things are logged when leaving a method (that is, the **capture** method is called as a postamble).

- Method name
- Name of the class in which the method is defined (it could be a superclass of the receiver)
- Return value (type and value)

In addition, extra information is logged, based on the type of the receiver; for example:

- For all controls, show the parent, index, and persistentObject
- For tables, show the row, column, inputType, and so on
- For folders, show the top sheet
- For buttons, show the caption
- Show bubble help, if it is not null

Nested method calls are shown with indentation.

This diagnostic information can help when trying to determine if something else needs replay code generated for it. It can also help identify what happened and when, including in situations when you cannot remember what you did while recording. The **rrmsg.log** is therefore invaluable when resolving problems.

Parsing Tokens

Code fragments (snippets) are in **rrmsg.log** lines prefixed with parsing tokens. The rest of the line is the generated code fragment. Multiple-line fragments are punctuated with **\n** where line feeds should be.

Some of the more-common parsing tokens are listed in the following table.

Token	Description
methodName=	Contains the name stem of a new method, usually of the format *form-name-acronym_ method-name*; for example, **methodName=CN_lstTest_dblClick**. The final method name also includes a prefix with an acronym of the generated class name, a sequence number, and a variant indicating whether it is a validation or a modal method. This causes a method break; any subsequent code goes into the next method. This is usually generated for all **click** and **dblClick** event methods, as these actions are most likely to cause transactions, which require the pause afterwards for the transaction to take place.
methodSource=	Source of the new method, matched one-for-one with the **methodName=** token. These can be multiple line, using **\n** for line feeds. This source forms the end of the method. Any saved source is inserted at the top of this method.
methodSaveSource=	Source to be included in the next method; that is, the code is needed, but a method break is not required.
referenceName=	Reference names to be included as class attributes.
methodVars=	Method variables to be included at the top of the next method. These are most commonly used with **getPropertyValue** or **getControl** methods, to hold a reference to a control that was not painted on an inherited form.
startModal=	A modal form was loaded.

Code Generation Methods at Recording Time

Some important methods that are called from the **capture** method are listed in the following table.

Method	Description
genMethodLocalVars	For peer schema access or dynamic controls, generates **methodVars=** for local variables and **methodSaveSource=** containing **getPropertyValue**, **getControl**, and so on.
getFullControlName	Returns the control name as required in generated code, taking into account peer schema access, generated form reference names, and so on.

Method	Description
indentOk	Facilitates suppression of code generation for tracked **inheritMethod** calls. It is generally called with **elseif …** and **indentOk(app.atcgIndentMax) then…** Note that there are other, older mechanisms for handling this, which are being phased out. The **checkIndent** method lets you comment out lines if they are indented too far. In **parseInputFile**, indentation can be directly checked and code generation suppressed. The **indentOk** mechanism should be used where possible. In some cases, **isInheritMethod** can be used to explicitly check for **inheritMethod** calls, but we usually want to suppress code generation when application code calls **Button** control **click** event methods, for example. These will show as indented but not inherited, so the **indentOk** method is usually the correct one to call.
isPeerSchemaClass, controlWasAdded, and controlWasLoaded	These methods facilitate checking for special code generation requirements.

Parsing and Code Generation

At the end of a recording session, you click the **Generate and Load** button. The **rrmsg.log** is read in by the **parseInputFile** method, extracting code fragments based on the parsing tokens described in "Parsing Tokens", earlier in this chapter.

The code fragments are saved to a number of arrays and flat files. The arrays and files are then combined into a .**cls** file by the **genClassFile** method . The .**cls** file is then loaded into the database using **jadloadb**, the batch Schema Load utility. All of the files involved are in the **logs** folder for the database.

In general, it is best to avoid complexity in the **parseInputFile** method. Its processing is of necessity quite convoluted, with some code fragments needing to be attached at the beginning of the prior method, or called from the **handleShowModal** method instead of the **runTest** method, and so on. The complexity is much more easily handled at capture time, when all of the original objects are easily accessible. If something extra needs to be generated, take care to minimize changes to the **parseInputFile** method.

Note The fact that the .**cls** file contains **jadeVersionNumber "6.2.15"** indicates that it is not intended to be replayed on a database with a JADE version older than **6.2.15**. It is *not* an indication of the level at which it was recorded.

Replaying Actions

After loading the profile class, click the **Replay Last** button. An instance of the newly-generated class is then created and its **startup** and **runTest** methods are called.

If a list is being replayed, the **startApplication** method is called for each profile class in turn. Each profile class must therefore assume that only the standard forms are open when recording starts. The required state of the persistent database is under your control; for example, you can create an object in **Test1** that is used in **Test2**.

The **AtcgReplayApp** application should work in standard client, presentation client, or single user mode. The **AtcgReplayApp** application itself does not currently permit multiple copies running, but you can easily write a harness that supports this. The essential code is as follows.

```
vars
    tran: AtcgProfile;
begin
    app.initialize;
    create tran as name-of-your-transient-profile-class transient;
    tran.startup;
    tran.runTest;
    tran.stop;
epilog
    delete tran;
end;
```

The class names could be read in from a flat file, for example, and looked up with something like a **currentSchema.getClass(className)** method.

The **app.initialize** method call can vary, according to the requirements of your tested application.

Chapter 4 Design Guidelines and Directions

This chapter covers the following topics.

Accuracy of Replay

At the highest level, the intention is to capture control event method executions (to determine what GUI actions were taken) and generate replay code to repeat those actions.

The general approach is to track and log all control and form event methods and application (**app**) methods, and generate replay code for them as it is determined that it is required. Normally, we just want to replay the RootSchema method, not reimplementations; for example:

```
ListBox.click(CMUI_NavigatorListBox2/21014.1);
    CMUI_Navigator.lstTest_click(CMUI_NavigatorListBox2/21014.1);
    CMUI_Navigator.lstTest_click:null
ListBox.click:null
```

We want to replay the **ListBox.click** event method and not replay **CMUI_Navigator.lstTest_click**. As **ListBox.click** will call **CMUI_Navigator.lstTest_click** at replay time, calling them both would double up the action.

There are issues with simply replaying the event methods that are called, as follows.

- Most of the event methods do nothing.
- There is not a one-to-one correspondence between event methods and the original GUI actions.
- There are GUI actions for which no events are generated.

  In addition, the application can actively suppress any or all event method calls, making it difficult for the recording process to determine what GUI actions have occurred.

- Calling an event method programmatically is not the same as performing the GUI action. For example, calling the **Table** class **click** event method does not do the following things, while clicking in the table does.
  - Set focus to the table
  - Set the **selected** property value
  - Change **row** and **column** property values

In addition, a table can behave differently, depending on whether it has focus. For example, programmatically calling the **Table** class **click** event method for a cell with the **cellControl** property set to **table** (that is, a table within a table) does not expand the cell and show the table, but clicking in the cell does.

Another example is that calling the **gotFocus** event method does not accomplish the same thing as calling the **setFocus** method, yet it is the **gotFocus** event method that is tracked and from which code is generated.

Awareness of State while Recording

Sometimes we need to be aware of various state information while recording. For example, with a table cell that has the **inputType** property set to **InputType_ComboBox**, when you leave the cell, the text is copied from the combo box text to the table cell text. This does not occur automatically just by replaying all of the control events, so we need to set the text explicitly. The logical time to do that would be triggered by the **queryRowColChg** event method, as the old cell row, column, and text are still available. However, if the application suppresses **queryRowColChg**, we need to do it another way.

Another example is that it would be nice to rely on the **TextBox** class **lostFocus** event method to trigger the code generation to set the text box text, but if there is a **Label** class **click** method after entering the text, the **lostFocus** event method happens after the **click** method finishes. Since validation code is generated immediately following **click** events, the text has not yet been set and validation fails. There are numerous other examples.

The general technique for solving problems of this kind is to save off various state information; for example:

- If a **cellInputReady** event method occurs on a **Table** class, an **AtcgCellMate** is saved, recording the correspondence between the cell and the associated control. If the associated control is a text box, when a **lostFocus** event method occurs on the text box, all cell mates are checked to see if there is a table cell into which the text needs to be set.
- For each **keyUp** event method on a text box, the form, control, and text are saved. If a **click** event method occurs before the **lostFocus** event method on the text box, the code to set the text can be generated.

The fact that we are called back for each method individually means that we have to be careful about saving state. The receiver of the **capture** method (that is, **global.atcgMyControl**) is persistent, and we cannot begin or commit transactions without knowing the state of the tested application (for example, the commit would release most locks). This means we need to save everything off on **app**, or a transient referenced from **app**. At the moment, we keep these items on **app**, but if there get to be too many of them, we could invent a new helper class and use a single reference to it on **app**.

The **AtcgCellMate** class was originally implemented for controls associated with table cells (hence its name), but has since been expanded for general-purpose state saving.

Performance

To support the replaying of GUI actions in thin client mode, especially for benchmark use but also for regression testing, we don't want to make large numbers of unnecessary method calls that would require network traffic.

The performance of the **AtcgRecordApp** application is important, as we do not want to significantly slow down the recording process if we can possibly help it. In addition to making it more difficult to use, it could introduce timing issues in the logic of the tested application. You should therefore eliminate voluminous logging of calls to methods on **app**, by adding the method to the **noiseMethods** array in the **atcgGetControlOptions** method of the **Application** subclass.

The two methods that write to the **rrmsg.log** are the **Application** subclass **atcgLogMessage** and **atcgLogInfoMessage** methods. The **atcgLogMessage** method uses **JadeLog.log**, which waits for each write to complete. The **atcgLogInfoMessage** method uses **JadeLog.info**, which buffers the writes.

Tip Because of the volume of writes, you should use the **Application** subclass **atcgLogInfoMessage** method whenever possible.

To ensure that the buffer is flushed periodically, the **capture** method uses the **atcgLogMessage** method for the postamble of methods when the value of the **Application** class **atcgIndent** property is set to zero (**0**).

Code Maintenance

In general, you should do all data manipulation for code generation at capture time; not at parsing time. At capture time, all of the original objects are easily available. The alternative is to generate intermediate tokens and write parsing code for them. It is easier just to generate the correct code to begin with, especially if attributes of several objects need to be taken into consideration to make a decision (for example, peer schemas and dynamic controls). It also makes the **rrmsg.log** much easier to deal with.

As the **capture** method is necessarily complex, take care to keep it clean and tidy. Where practical, you should transfer larger chunks of code to additional methods, to keep the size of the **capture** method under control. However, this needs to be balanced against creating a large number of methods in the class, which would also be unwieldy.

To help avoid a naming conflict when ATCG schemas are loaded under an application schema, some naming conventions need to be strictly adhered to.

- In the **AtcgGeneratorSchema** schema, all class names other than those based on the schema name (that is, **app** and **global**) must begin with **Atcg**.
- All methods and properties on **app**, **global**, **JadeScript**, and **RootSchema** classes must begin with **atcg**.

  This also applies to all primitive methods, external methods and external functions. It does not apply to method reimplementations on app such as app.msgBox.
- In the **AtcgTestCodeSchema** schema, the same rules as those for **AtcgGeneratorSchema** apply for classes and methods that "belong" to ATCG.
- For methods that are designed to be modified for use with different databases (for example, **app.atcgGetControlOptions**), there is a second copy of the method with the suffix **Sample**; for example, **app.atcgGetControlOptionsSample**.

Ease of Use for the End User

The generated code should be as easy as possible for your end-users to modify, which means that you should:

- Use names that are meaningful
- Apply indentation to make it easy to read
- Define a structure that is easy to understand

Finding References at Replay Time

One of the challenges in ATCG is finding the right references at replay time that correspond with the original controls at record time. For example, while recording, we find out about a new form when the **Form** class **load** event method is called. The reference to the form is the receiver of the method, so there is no mistake. However, at replay time, there may be several forms available from **app.getForm** with the same name, so we need to get the right one.

Dynamic controls is another example. The **Control** class **loadControl** method allows a control (usually a painted control) to be cloned and added to the form. The **Form** class **addControl** method allows any number of dynamically created controls to be added to the form. These new controls are created from scratch on the fly, and they may bear no resemblance to any existing control. Toolbars and navigation panes are frequently made up of these dynamic controls.

In the case of the form references, ATCG retrieves the references as the forms are opened, so they are kept straight. The **app.getForm** method returns the most recently opened form.

For references to controls that are loaded from painted controls, those are retrieved at replay time using the **Control** class **getControl** method. The correct index to use is captured at recording time.

There is no way to be certain about getting the correct references to controls that are added or those that are loaded from added controls. The current technique is to match them based on the following criteria of the control.

- **bubbleHelp** text
- **name** property value
- Class name
- **left** property value
- For left-aligned controls, the distance to the right edge of the parent
- **top** property value
- **index** property value

These criteria are all checked for the control, its parent control, the parent of its parent control, and so on, up to the form.

To allow for right-aligned controls (for example, toolbars) where the form has been resized, it is considered a match if the value of the **left** property or the distance to the right edge of the parent matches.

Modal Forms

The **Form** class **showModal** method blocks execution of code until the form is unloaded, so actions cannot be replayed on a modal form.

To handle this, a **handleShowModal** method is generated in the profile class. It calls the **Form** class **show** method rather than the **Form** class **showModal** method. Profile methods can then be replayed as normal. At the end of the **handleShowModal** method, the value that was captured at recording time is returned.

To support this, you must reimplement the **Form** class **showModal** method to call the **handleShowModal** method. If the application has reimplemented this in its common superclass, you can change that reimplementation.

Nested modal forms are supported.

There are several parsing tokens specifically for modal forms. It is necessary to strictly separate the code that is generated for action on the modal form versus that for the other forms; for example, a single reference to the modal form after it is closed generates an exception.

Common Dialogs

Uses of selected **CMDialog** subclasses are supported. As these common dialogs are modal and cannot be reimplemented, they need to be handled differently.

At replay time, the necessary **CMDialog** subclass methods are tracked, which call back the **AtcgControl** class **replay** method to take the actions required. For example, if the **CMDColor** class is used, when the **CMDColor** class **open** method calls back the **AtcgControl** class **replay** method, the **AtcgBtnClick** application is started during the preamble, which is told which button to click. It does this by calling Windows APIs. During the postamble, the recorded color is set in the **CMDColor** object. The recorded color was stored in a property in the **Application** subclass, which was set before the dialog was brought up.

The **CMDFont** and **CMDPrint** classes are not currently supported.

User Hook for Dynamic Text

In some cases, dynamic text needs to be handled; for example, a Personal Identification Number (PIN) code is displayed in a message box and needs to be entered into a text box later on. The PIN code that is displayed at replay time is the one that needs to be entered into the text box at replay time; the PIN code that was displayed at recording time will not work.

To handle this, a special click was defined for **TextBox** controls; that is, Ctrl+Shift+left-click. This identifies the text box as dynamic. At replay time, this means that the text in the text box is set from the return string of **app.atcgDynamicContent** rather than the string that was entered at recording time. You can then reimplement **app.msgBox** to save off the required text, and modify **app.atcgDynamicContent** to return it at the correct time.

Application Message Box

When message boxes occur at replay time, the same action needs to be taken that was taken at recording time. To handle this, the return code is captured at postamble time along with the message box title, message string, and flags. When this combination of parameters is encountered at replay time, the recorded return code is returned.

The **runTest** method in the profile class declares what message boxes are expected during execution of the class. These declarations are checked at replay time in the reimplemented **app.msgBox** method. If unexpected message boxes are displayed or expected message boxes are not displayed, the replay fails.

To allow for dynamic content (for example, "Are you sure you want to delete user John Smith?"), a match of the flags is considered close enough, if the message box is displayed at the expected time. A message is logged, showing the expected and actual parameters involved. You could make this algorithm more sophisticated; for example, requiring that the beginning or end of the title or message, or both the title and the message, must also match.

List Box and Combo Box Actions

To allow for changes in the tested application between releases, lines are found at replay time based on text rather than index. This allows for a new release of the application inserting an additional item into a navigation list box, for example.

For hierarchical list boxes, the text at each level is recorded and used.

Multiple Applications Support

To support multiple applications with the same **AtcgGeneratorSchema** schema, some options are passed in using **app.atcgGetControlOptions** method in the **AtcgTestCodeSchema** schema. This includes:

- Generated schema name
- Generated superschema name
- Generated superclass name
- Name of the ***logMessage*** method of the driver schema
- List of schemas to be tracked
- List of noise methods to exclude from method tracking

Note The **global.atcgMyControl** method in the **GAtcgGeneratorSchema** class can be accessed from the **atcgGetControlOptions** method, so you can actually set other options that are not provided for with explicit parameters.

Coexistence with Benchmark Driver Code

ATCG code and benchmark code can coexist in the **AtcgTestCodeSchema** schema.

The profile class structure, start-up infrastructure, and so on, are compatible. To help keep them straight at run time, the following flags properties are provided in the **AtcgGeneratorSchema** subclass of the **Application** class in the **AtcgGeneratorSchema**.

- atcgRunningAtcg
- atcgRunningAtcgReplay
- runningBenchmark