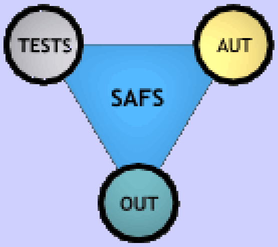
# SAFS101: Introduction to SAFS

## The SAFS Framework

This tutorial will introduce the student to the SAFS Framework. We will see how the framework provides these key benefits:



- Separate Test Designer and Automator roles
- Separate Test Designs from automation tools
- A shared testing framework for all platforms

In the lesson topics below we will actually provide separate content paths for the Test Designer (**TD**) and the Test Automator (**TA**) roles. Of course, you are free to explore both paths to get the full breadth of information.

You can start things off by telling us your current level of SAFS experience in the quiz/survey listed below.

SAFS Terms

Your SAFS Experience To Date

# SAFS Terms

## A

**App Map:** The nickname for an Application Map.

**Application Map:**

An Application Map is our centralized, single-point of maintenance storage mechanism for Window and Component definitions -- also called recognition strings.  Application Maps are opened and processed by SAFSMAPS to make them globally available to all testing tools in the environment.

The Application Map is where we "map" simple user-defined component names to the complex recognition information needed by the underlying automation tools.  This is also where we can store predefined Application Constants that will be referenced by the SAFSVARS variable service when necessary.

"App Map" is a common nickname referring to an Application Map.

**Application-Independent:** The framework or tool is not exclusively designed to work with just one application.  It is intended to work across many applications, or all applications, without modification. Simply, it was made to test all applications, not just one.

**AUT:**

**Application Under Test**

The application we are testing.

**Automator:** See "Test Automator"

## D

**Designer:** See "Test Designer"

**Driver:**

The *Driver* is generally the tool, class, or entity that is responsible for the overall execution of the test. The Driver opens and reads our test tables, keeps track of test status and results, and performs the parsing and routing of test records to the various Engines for execution.

Some tools -- like RRAFS and WRAFS -- are standalone execution "Engines".  They provide both the Driver and the Engine for execution.  Other tools -- like SAFSDRIVER -- are strictly Drivers and require the use of external Engines to complete a fully functional testing framework.

## E

**Engine:**

In a pure sense, an Engine is used to execute the instructions retrieved by the controlling test Driver. For example: the "RobotJ Engine" refers to the Engine developed for Rational RobotJ (XDE Tester). This is strictly an "Engine" that requires an external Driver to function.

However, the term Engine often refers to a complete testing framework including both Driver and Engine. For example: the "RRAFS Engine" refers to the complete package of Driver and Engine developed for Rational Robot. Similarly, the "WRAFS Engine" refers to the complete package of Driver and Engine developed for Mercury WinRunner. Both of these implementation have inseparable Driver and Engine components.

## F

**Functional Scripting:** The test technique in which automation tool scripts, or script libraries, are broken down into callable functions or subroutines. These functions provide simple reusable actions or features that can be called many times by external scripts or script libraries, usually with different parameter values to provide variation on the function outcome.

**Functional Tester:**

This is the "will be known as" name for what used to be called Rational RobotJ, and is currently called IBM Rational XDE Tester. This is an Eclipse-based functional test automation tool for Java and Web environments sold by IBM Rational Software.

See: RobotJ for more information.

## I

**Input Record:** See: "Test Record"

## K

**Keyword-Driven:** A test technique in which tests are expressed with user-defined keywords and actions. A test interpretter is coded or trained to evaluate these user-defined expressions and perform the desired activities. These tests are usually written external to the testing tool. For example, they may be in text files, spreadsheets, or database tables.

## R

**RobotJ:**

This is the "formerly known as" name for what is now called IBM Rational XDE Tester, or IBM Rational Functional Tester. This is an Eclipse-based functional test automation tool for Java and Web environments sold by IBM Rational Software.

This is also the nickname of the SAFS Engine developed for this tool. The RobotJ engine is the first "pure" engine developed with the SAFS Framework. It has no Driver and, thus, requires some external Driver to handle the overall test. Rational Robot is usually the Driver used in conjunction with the RobotJ engine.

**RRAFS:**

**Rational Robot Automation Framework Support**

Or more correctly: the SAFS Engine for Rational Robot. This was the first, and is the most complete keyword-driven automation framework offered by SAFSDEV. It is considered to be the "reference implementation" for all SAFS Engines. It is a standalone Engine containing both Driver and Engine implementations. However, RRAFS is also capable of being the controlling Driver calling multiple external Engines.

Often, the term RRAFS is intended to infer the inclusion of Rational RobotJ or Rational XDE Tester -- which is a separate SAFS Engine -- but often readily available to Rational Robot users.

## S

**SAFS:**

**SAFS -- Software Automation Framework Support**

This is a collection of multi-platform tools and protocols designed to facilitate software test automation. The intended focus of these tools is to augment test automation across many different automation tools. It also defines standard processes and protocols for disparate tools to share data and information.

The SAFS tools -- often referred to as the SAFS Framework -- are used throughout the keyword-driven test automation frameworks developed and released by SAFSDEV on SourceForge at http://safsdev.sourceforge.net

See also Driver, Engine, and SAFS Engine.

**SAFS Engine:** See "Engine"

## S (cont'd)

**SAFSINPUT:** SAFSINPUT is the STAF service provided by the SAFS Framework to handle machine-global file input. The files opened in SAFSINPUT are available to all STAF-enabled testing tools in the environment. However, generally, these files are most often the test tables opened by the active Driver in the environment.

**SAFSLOGS:** SAFSLOGS is the STAF service provided by the SAFS Framework to handle machine-global test logging. Initialized logs in SAFSLOGS are available to all STAF-enabled testing tools in the environment. All testing tools are expected to log to this service so that the sum of all messages from all tools forms a single test log.

**SAFSMAPS:** SAFSMAPS is the STAF service provided by the SAFS Framework to handle machine-global Application Maps. The Application Map data and Application Constants stored in SAFSMAPS are available to all STAF-enabled testing tools in the environment. This service can work in conjunction with SAFSVARS to provide those Application Constants as available variables.

**SAFSVARS:** SAFSVARS is the STAF service provided by the SAFS Framework to handle machine-global variables. The variable data stored in SAFSVARS is available to all STAF-enabled testing tools in the environment. This service can work in conjunction with SAFSMAPS where the user can specify predefined variable values often called "constants" or "application constants".

**STAF:**

**STAF -- Software Testing Automation Framework**

STAF is an open source test automation framework managed by IBM designed primarily for distributed testing across multiple machines and platforms. STAF provides enormously powerful automation utility via an ultra-simplistic interface.

This simple interface is accessible to many different programming languages on many different platforms. It allows all of these disparate environments to communicate, coordinate, and synchronize test automation in a distributed environment.

Visit the STAF Homepage on SourceForge.

## T

**Test Automator:**

The test automator develops and maintains successful automation based on the test designs provided by test designers, and the automation tools used to execute the tests.  Designing tests generally consumes more time than automating, so test automators can usually support multiple test designers.

**Test Design:**

A test design generally contains the flow of desired test progression, user or system input, and an expected result where applicable.  One might say that a test design is "a test"; and test designs are "the collection of tests".

In SAFS these are also often called "Test Tables".

**Test Designer:** A test designer generally defines or creates test designs--the tests.  They focus on the scenarios for testing the application, and expressing those in test designs.  Test Designers are not concerned with automation tool details, languages, or complexities.  That is the role of the Test Automator.

**Test Framework:** The collection of associated tools, technologies, and predefined processes deployed collectively as a system to accomplish test automation.

**Test Record:**

Loosely, a test record is an individual "line" or "row" of a test table.  It generally contains a single command or action to execute.  This is also known as an "input record".

Strictly, this is a specific *record type* in a test table designated in field #1 by the value of " T ", " TW ", or " TF ".  The success or failure of this record type increments "test" pass/fail counters instead of "general" pass/fail counters.

**Test Table:**

A SAFS test table generally contains the flow of desired test progression, user or system input, and an expected result where applicable.  These are the test records that define what is to be done.

One might say that a test table is "a test"; and test tables are "the collection of tests".

Generically also called a "Test Design".

**Test Technique:**

The physical methods by which test automation is expressed or captured.  The most common of these would be *Record and Playback*, *Functional Scripting*, and *Keyword-Driven* methods.

**Tester:** See "Test Designer"

## W

**WRAFS:**

**WinRunner Automation Framework Support**

More correctly: the SAFS Engine for WinRunner. This was the second automation framework offered by SAFSDEV. It was a straight translation of the Rational Robot engine into WinRunner as provided by John Crunk. It is a standalone SAFS Engine containing inseparable Driver and Engine implementations.

## X

**XDE Tester:**

This is the "also known as" name for what used to be called Rational RobotJ, and may soon be called IBM Rational Functional Tester. This is an Eclipse-based functional test automation tool for Java and Web environments sold by IBM Rational Software.

See: RobotJ for more information.

## Survey: Your SAFS Experience To Date

Don't worry! There are no wrong answers for this quiz. We just want to get an idea of your current level of exposure to the SAFS Framework. Don't worry if you are unfamiliar with these terms. You will learn about these in the lessons, or you can review the Glossary to find the definitions for these terms.

**1** Rate your use, or intended use of RRAFS--the SAFS Engine for Rational Robot.

Answer:
○ a. Not planning to use it.
○ b. Never used it and know little or nothing about it.
○ c. Read about it and investigating further.
○ d. Installed it and want to learn how to use it.
○ e. Used it some and want to know more.
○ f. Very actively using it.

**2** Rate your use, or intended use of WRAFS--the SAFS Engine for Mercury Interactive WinRunner.

Answer:
○ a. Not planning to use it.
○ b. Never used it and know little or nothing about it.
○ c. Read about it and investigating further.
○ d. Installed it and want to learn how to use it.
○ e. Used it some and want to know more.
○ f. Very actively using it.

**3** Rate your use, or intended use of SAFS/RobotJ--the SAFS Engine for Rational RobotJ/XDE Tester/Functional Tester.

Answer:
○ a. Not planning to use it.
○ b. Never used it and know little or nothing about it.
○ c. Read about it and investigating further.
○ d. Installed it and want to learn how to use it.
○ e. Used it some and want to know more.
○ f. Very actively using it.

**4** Have you ever used the STAF-enabled SAFS Framework tools like SAFSVARS, SAFSMAPS, and SAFSLOGS in conjunction with any SAFS Engine?

Answer:
- ○ a. I haven't used any tools or engines just yet.
- ○ b. I have used RRAFS, but not the STAF-enabled tools.
- ○ c. I have used WRAFS, but not the STAF-enabled tools.
- ○ d. I don't know if I have used the STAF-enabled tools.
- ○ e. Yes. I have used these STAF-enabled tools.

This Page Intentionally Blank

**Lesson**

**1**

### The Test Roles for Thee

With SAFS we can separate the two main test development roles: the Test Designer, and the Test Automator.

Which role tolls for thee?

Many roles: one purpose

## Many Roles: One Purpose

### Why separate?

The question does arise -- how and why do we separate the roles in Test Automation? We're really glad you asked!

First off, virtually every line of work separates roles. Doctors' offices employ nurses, secretaries, consultants, managers, pharmacists, assistants and clerks. Auto mechanics may have a receptionist, parts specialist, transmission specialist, and "the engine guy".

Software developers often separate code responsibilities based on developer specialties and sometimes the sheer volume of code needing to be developed. So when it comes to testing the software, can we really expect one person, or one type of person to do it all? After all, the same volume of code must be tested -- and then some.

Separation as it pertains to testing in SAFS generally falls into two categories: Designers and Automators. The framwork itself requires an additional role of Engine Specialist. This separation allows for a distinct delineation between outlining what needs to be done, and chasing down the details of how it gets done. It's more complicated than that, of course, but we shouldn't saddle broad thinkers (designers) with minutiae, and those that enjoy the intricate details of debugging and coding (automators) likewise may not be at their best in a designer role.

Separation allows everyone to contribute what they are best suited to give. It allows for specialization and the concentration of effort and expertise where it can best be used.

## Advantages of separation

When workers are allowed to focus on a particular task or specialty, the opportunity to excel rises. Without the distraction of tasks best suited for someone else, a worker can better focus on particular tasks that meet one's natural skills. This typically results in less frustration for workers since they are able to do things they enjoy, are good at, and have a clearly defined responsibility for.

Project managers will also see advantages, since areas of responsibility are clearly marked and there is likely to be less overlap and less rework. With the familiarity of the focus area, an adept automator can reuse previous work and make small changes to keep large projects in sync.

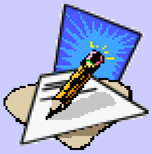
## Role of the Designer

The Designer in our keyword-driven approach has a vital task that is not typically found in other testing models. They don't really test like ad-hoc manual testers, though there could be some overlap. They have almost nothing to do with a record/playback methodology, though their output will be run time and time again. They might think in terms of *scripts*, but aren't likely to ever write or implement one in the traditional sense. And designers don't need to have any knowledge of the specific testing tools used (like SAFS), even though these tools will be key in automating their design output.

So what do designers do? And how do they differ from just a plain ol' "tester"? 😕

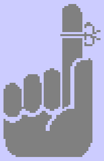
While a designer may be involved in tasks that bridge a wide part of the software development landscape, the group that may best define the focus of the designer is the Customer. Did we say the Customer? Why yes, we did. And it sounds so obvious, but let's explain this a bit further.

## Defining a Tester's Job



There has been a resurgence in the focus of marketing in the software industry to focus a product offering on what the customer wants. (Granted, this seems far-fetched, but work with us here.) This has resulted in development teams being sent to the field. The formation of focus groups, customer surveys, and customer labs both inside and outside of the company. But in all of this we must stop and ask the very important question:

*Just what is the tester's job, anyway?*



A growing number of professional testers believe that a tester should efficiently prove that the software works as intended. And, of course, this includes efficiently proving the software does not do what it is not intended to do. These intentions are specified in product requirements.

So do we separate the tester from the development, marketing, and even customer support elements of the company? The answer is, no we shouldn't.

A designer is this proverbial tester and thinks like a customer. The designer focuses on what the software *should* do based on the requirements formed from customer input, marketing, development, and often the designer's own input and then strives to answer the oft-repeated question:

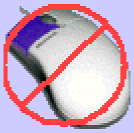
*What are the user scenarios that verify this?*

## Defining the User Scenarios

So how do we define these user scenarios?

The answer is surprisingly simple. Yet often times so surprisingly difficult for "old school" testers to adjust to, initially. The concepts of "How will this be done?" often cloud the analysis of "What needs to be done?", and that is a problem. We need the designer to tell us what needs to be done in an abstract, customer-centric manner and not be concerned with the details of how it will be done.

| | | |
|---|---|---|
| Login | ^user="jones" | ^pwd="meandmrs" |
| StartNewUser | ^name="John Jacob" | ^id="jingleheimer" |
| AddUserAddress | ^add1="123 JJJS Dr." | ^city="Zebulon" |
| SubmitNewUser | | |



As shown above, the designer must focus on abstract user scenarios and not the testing tools used or the GUI actions involved (if any). If a tester/designer were to focus on GUI navigation or ordered GUI actions then the test can only be run this way. What if we also want to run the test against the application API and bypass the GUI? And what if we don't even have an application GUI yet? Must we wait for a GUI before the designer can build these tests? (The answer to that one, of course, is "no".)

"What needs to be done?" is the realm of the Test Designer. "How will this be done?" is the separate realm of the Test Automator. The subtlety of this difference may slip by you -- please don't let it.

## So what does a designer do?

A designer's role is to convert the requirements of the system -- what the software is intended to do -- into a workable test design that can prove that those requirements have, in fact, been satsified. While traditional "testers" often perform functions similiar to or overlapping with a dedicated test designer, we find that most testers do not think like a designer, but rather...well...an application pounder. It's a subtle difference that makes all the difference when transitioning to a keyword-driven approach.

A designer doesn't need to see the GUI -- a working product isn't even required! Many people even think it's better to work on designs *without* seeing the end product. This ordered approach -- written in (fairly) plain language -- is known as the Test Design, and it is composed of abstract user scenarios that become the basis for the keyword-driven tests that prove our requirements.

**Quesion: Which of these things would a designer need to focus on?**

○ User feedback
○ Requirements document
○ GUI elements
○ All of the above

## Designer Example

Let's take a look at an example of how a designer may work.

Marketing has identified the need for a product that tracks credit card usage. As one of the requirements for security, the product requires logon credentials from a user whose account exists in the company's domain. If the user does not have any activity in the application for 10 minutes, further work in the application is prohibited without another logon. Logging off manually has the same effect.

The designer looks at these requirements and identifies some major issues that must be tested, namely:

- incorrect or missing credentials prevents application access
- valid logon allows access
- logoff prevents further acccess without valid logon
- timeout prevents further access without valid logon

The designer then asks, *What are the user scenarios that verify these?*
The working test design could eventually look something like this:

```
Start Application
Enable logon process
Provide valid logon credentials
Verify access granted
Logoff
Enable logon process
Provide invalid logon
credentials
Verify access denied
Provide valid logon credentials
Wait timeout period
Verify access denied
Provide valid logon credentials
Verify access granted
```

The table above is referenced in the next few pages as the “previous test design”.

## Defining Reusable Actions

From the previous test design, the quick reader will note that we really are only doing a few tasks, but with different parameters. First, we must crank up the application. Next -- and we do this several times and in several ways -- we logon (or attempt to logon) to the application. Then we test that a logon (successful or not) leaves the application user in the proper state. Throw in a couple of tasks that don't fit the basic flow like logging off and waiting for a while and the entire testing process of this requirement can be summed up with these few "Actions".

Once defined, these Actions are *not* limited to use in the logon scripts. These can be part of other scripts to develop and create even more testing possibilities unrelated to application security. (After all, we must always logon to the application before we can test its other features.) If some part of logon should break, fixing it in one oft-used logon Action fixes it in every script that includes that logon Action.

Note some differences between this approach and your typical manual testing. We are *not* necessarily "looking for bugs". We are *not* just whacking at the program trying to make it break (that comes later ☺ ). We *are* methodically defining actions that verify the application or system satisfies our requirements.

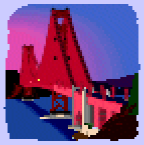
Note also that this logon design will work with *any* such application, and *any* test automation methodology. We don't know if it's a web application, a standalone app, the target OS, what database or technology/protocol is being used, or whether encryption is enabled -- and we don't care.

What we do care about is that we have a simple description of required security testing and logon activities that could be read by anyone. And anyone having a decent knowledge of the application could take this description and actually perform the testing.

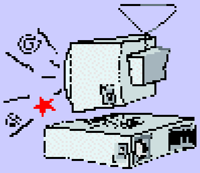
## Role of the Automator

The MOTO (Master-Of-The-Obvious) definition of Automator would be "one who automates." And while that's exactly what our Automator does, we'd like to extend that definition to include answering the original test design question of "How is this going to be done?". It's worth noting here that a well-written test design can be snatched up and used for manual testing by a human -- almost any human (even developers) -- without any intervention of a Test Automator.

A well-written test design should be easy to understand and anyone with a working knowledge of how to use the product should be able to take the descriptions and directives of the design and execute them. These same tests bridge into the domain of the Automator. The automator will take the abstractions of the test design and make them work in the concrete context of the automated testing framework. With that in mind, we have identified our first skill required for the Automator -- a knowledge of test automation tools and frameworks.



A good Automator may get recognized for a seemingly complex task and a well-written automation implementation. Yet only those that Automate need to know how easy SAFS makes it. If the test is primarily clicking and typing, the SAFS framework provides trivial pre-implemented actions for those tasks and the job of the Automator is little more than scripting those events to happen.



But good automated tests are more than simple GUI execution. It's flow control and scripting or programming to handle the things that break traditional test automation scripts. Let's start by looking at this programming element to the Automator's role.

## Automator Example

Review our earlier Test Design example and see what that looks like in the world of the Automator. First, our Designer identified some Actions and handed them to the Automator to implement. It's the Automator's job to make those actions "work". Let's take a peek at the Logon action as an example.

At first glance, this may seem trivial -- and it can be implemented that way. Logon is simple -- type in user name and password and click the button or pound the Enter key -- the harder the better.

| | | | |
|---|---|---|---|
| LoginWin | UserName | SetTextValue | ^user |
| LoginWin | Password | SetTextValue | ^pwd |
| LoginWin | Logon | Click | |

But there are more possibilities than meet the eye here. What if the user is already logged on and a logon attempt is initiated? What if this is a web application and the browser prompts to save the user name and password? What if ...??? These are the nightmares of an Automator. Any change or unexpected result can break unsuspecting tests (and Automators) easily. The automator must somehow account for and accomodate some of these possibilities.

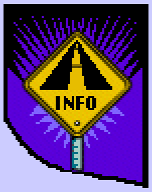
And that is key: Logon should be made into one highly reusable action invoked during many tests. If the application login requirements change, and our many tests relying on Logon begin to fail, we need to fix only this one Logon action for all affected tests to work again. And that is pretty cool!

**Q: Select 2 things considered part of the Automator's job?**

○ Program for unexpected occurences
○ Yell at Designers for poor Test Designs
○ All of the above
○ Transfer test designs into automated test Actions.

## The Engine Specialist

Slightly beyond the scope of this introduction but worth mentioning is the Engine Specialist. As described earlier, SAFS can support several automation backend tools (such as Rational Robot) to use for the execution of the GUI of the product. Currently, there are at least three working engines (Rational Robot, XDE Testers, and WinRunner). The role of the Engine Specialist is to create more engines and to extend the functionality of the existing engines.



The open source project of SAFS is primarily dependent on the work of volunteers to make the framework a robust and functional testing option. Work on a tool-independent driver is mostly completed. This will enable other operating systems and virtually any test tool -- present or future -- to leverage the capability of the SAFS Framework. Included in this development is the intention that existing test designs and their implementations will be able to use or migrate directly to these new engines.

If you'd like to be involved in the new development or ongoing support of SAFS engines, then you should monitor and post to the SAFS developers' forum. Basic programming skills and better-than-average knowledge of the programming language of the automation tool are required.

**Question: Who extends the functionality of the SAFS framework?**

○ The SAFSWorks staff
○ All of the above
○ The users
○ The dedicated development staff

**SAFS Developer's Forum**

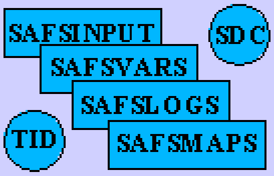
**http://lists.sourceforge.net/lists/listinfo/safsdev-developers**

This Page Intentionally Blank

**Lesson**

**2**

## Installing SAFS



The shared components of the SAFS Framework can be installed separate from any of the larger frameworks like RRAFS and WRAFS that normally include them. You can also upgrade the SAFS Framework components without upgrading those larger frameworks.

In this lesson we will perform an install of the SAFS Framework.

Who needs to install the SAFS Framework?

Performing a SAFS Install

Installation Graduation?

## Who Needs to Intall the SAFS Framework?

The SAFS Framework does not need to be installed everywhere. Only the machines that will actually *execute* the automated tests will need it. Thus, the Test Automator is typically the one that needs this install. If the Test Designer will not be executing the tests, then the Test Designer need not install the framework.

Additionally, a separate install of the SAFS Framework is not needed if a complete install of one of the larger frameworks like RRAFS is performed. These larger frameworks should install these tools for you when they support their use.

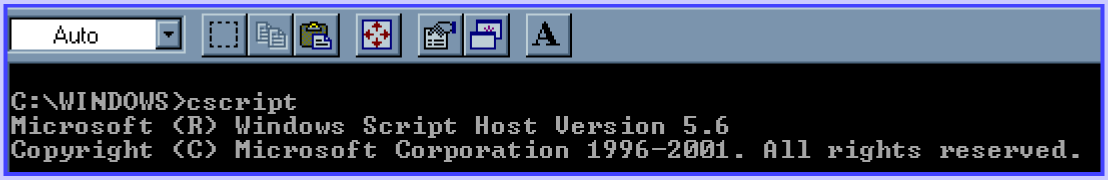
# Performing a SAFS Install

## SAFS Framework Prerequisites

An automated install on Windows requires:

1. **Windows Scripting Host 5.6**
   You can verify this version of WSH is installed on your system by typing the following at any command prompt and pressing ENTER:

   cscript

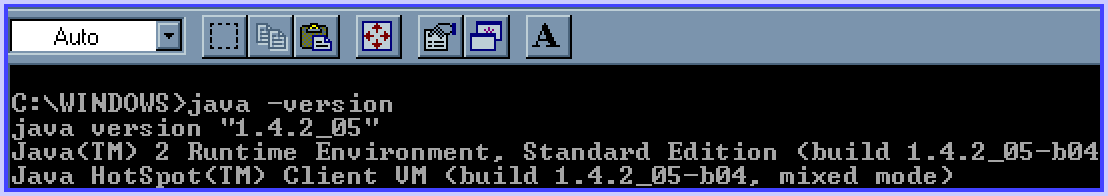


```
C:\WINDOWS>cscript
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001. All rights reserved.
```

   If the command does not correctly execute, or if the version is less than 5.6, you will need to install version 5.6 or later from the Microsoft Scripting Downloads website.

2. **Java Runtime 1.4**
   You can verify this version of Java is installed on your system by typing the following at any command prompt and pressing ENTER:

   java -version



```
C:\WINDOWS>java -version
java version "1.4.2_05"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2_05-b04
Java HotSpot(TM) Client VM (build 1.4.2_05-b04, mixed mode)
```

   If the command does not correctly execute, or if the version is less than 1.4, you will need to install version 1.4 or later from the Sun Java 2 Platform website.

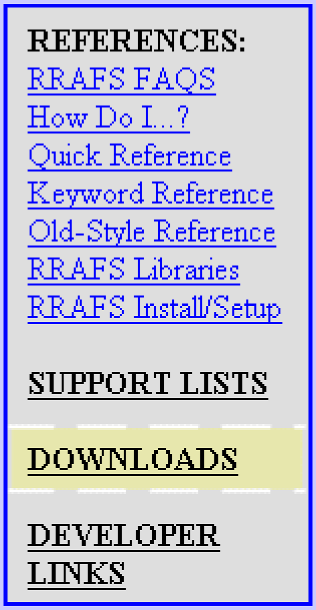
### Microsoft Scripting Downloads

**http://msdn.microsoft.com/downloads/list/webdev.asp**

### Sun Java 2 Platform

**http://java.sun.com/j2se/**

## Download the Latest SAFS Framework Release



**REFERENCES:**
[RRAFS FAQS]
[How Do I...?]
[Quick Reference]
[Keyword Reference]
[Old-Style Reference]
[RRAFS Libraries]
[RRAFS Install/Setup]

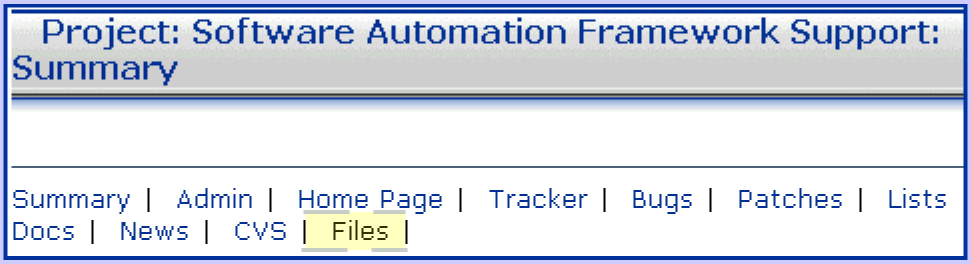
**SUPPORT LISTS**

**DOWNLOADS**

**DEVELOPER LINKS**

You can get to the SAFS Framework release thru the "**DOWNLOADS**" link highlighted on the left on the SAFS Home Page at http://safsdev.sourceforge.net.

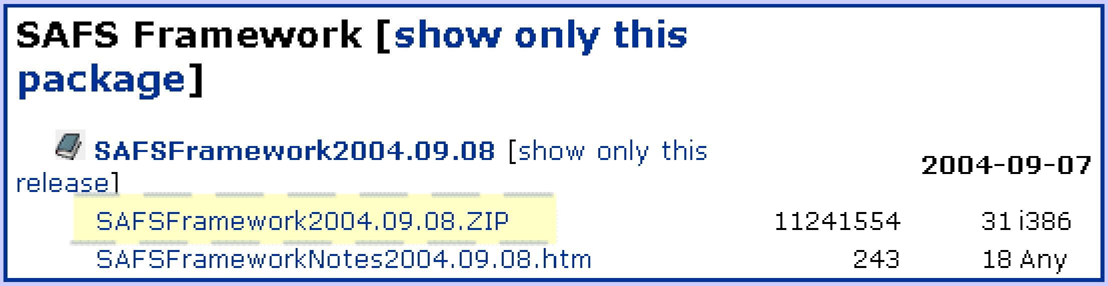
You can also get to the framework release thru the "**Files**" link highlighted below on the SAFS Project Page on SourceForge at http://sourceforge.net/projects/safsdev.



Project: Software Automation Framework Support: Summary

Summary | Admin | Home Page | Tracker | Bugs | Patches | Lists
Docs | News | CVS | Files |

Locate the latest *Release* from the "SAFS Framework" section of the Files page. (*Patches*, if present, can only be applied upon a previously installed release.) Review the release notes and then download the release **ZIP** file into a temporary location of your choice. For this tutorial we'll assume C:\TEMP.



**SAFS Framework [show only this package]**

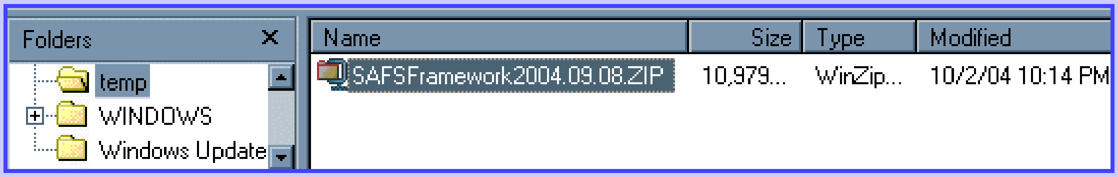
**SAFSFramework2004.09.08** [show only this release] **2004-09-07**

```
SAFSFramework2004.09.08.ZIP            11241554    31 i386
SAFSFrameworkNotes2004.09.08.htm            243    18 Any
```

**SAFS Framework**

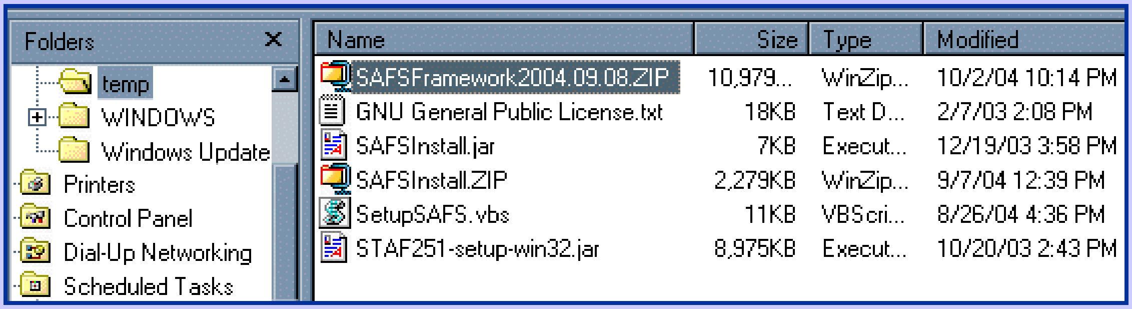
**http://sourceforge.net/projects/showfiles.php?group_id=56751&package_id=99532**

## Extract the SAFS Release Into Any Directory



| Name | Size | Type | Modified |
|---|---|---|---|
| SAFSFramework2004.09.08.ZIP | 10,979... | WinZip... | 10/2/04 10:14 PM |

Assuming the downloaded SAFS Release was placed in C:\TEMP, we must now extract the contents of the ZIP file into any directory. Well, there is no better place than that same C:\TEMP directory!



| Name | Size | Type | Modified |
|---|---|---|---|
| SAFSFramework2004.09.08.ZIP | 10,979... | WinZip... | 10/2/04 10:14 PM |
| GNU General Public License.txt | 18KB | Text D... | 2/7/03 2:08 PM |
| SAFSInstall.jar | 7KB | Execut... | 12/19/03 3:58 PM |
| SAFSInstall.ZIP | 2,279KB | WinZip... | 9/7/04 12:39 PM |
| SetupSAFS.vbs | 11KB | VBScri... | 8/26/04 4:36 PM |
| STAF251-setup-win32.jar | 8,975KB | Execut... | 10/20/03 2:43 PM |

## Review Custom Installation Options

The default SAFS Installer is actually comprised of 2 separate executables:

1. Java "*org.safs.installer.SilentInstaller*" class.
2. Windows Scripting Host "*SetupsSAFS.VBS*" script.

As with a standard install, all the assets extracted out of the release must be present in the directory that contains these installers.

Review the Java Command-Line Options. If you choose to install directly via this Java Installer you will have to invoke Java with this class and the arguments as parameter. For example:

- java org.safs.installer.SilentInstaller <installer args>

The problem with installing this way is that the default WSH installer sets required environment CLASSPATH and other variables for you. You will have to set these manually if you use the Java Installer.

- SAFSDIR=<root SAFS directory>
- STAFDIR=<root STAF directory>
- Add to CLASSPATH:
  - <root SAFS directory>\lib\SAFS.JAR
  - <root SAFS directory>\lib\SAFSCUST.JAR
  - <root SAFS directory>\lib\SAFSRATIONAL.JAR
  - <root SAFS directory>\lib\SAFSJREX.JAR
  - <root SAFS directory>\lib\JREX.JAR
  - <root SAFS directory>\lib\jakarta-regexp-1.3.JAR

Perhaps the better options is to call the SetupSAFS.VBS script on the next page! ☺

### Java Command-Line Options

**http://safsdev.sourceforge.net/doc/org/safs/installer/SilentInstaller.html#main(java.lang.String[])**

## SetupSAFS.VBS Custom Installation Options

Calling the Java Installer directly is an extreme case. So let's look at the VBS "SetupSAFS.VBS" instead.

Since our VBS actually calls the Java Installer -- the command-line arguments are the same. To invoke it:

- cscript SetupSAFS.VBS <install arguments>

Fortunately, this script knows how to set all the environment variables once the Java Install is complete!

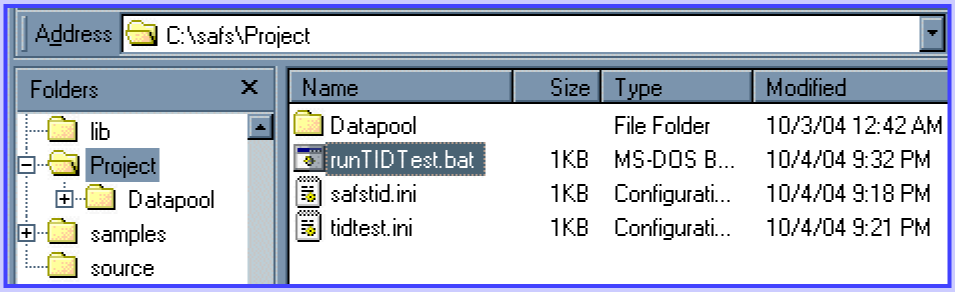
Finally, the next page will show us some simple housecleaning tasks we should do once the custom install is complete.

### Command-line arguments

**http://safsdev.sourceforge.net/doc/org/safs/installer/SilentInstaller.html#main(java.lang.String[])**

## Modify Installation Test Path Information

Once installed, the SAFS Framework has a simple little test that you can run to verify the SAFS Framework is working for you. However when you install SAFS to a custom directory the path information embedded in the test configuration files must be modified to match your custom install path.

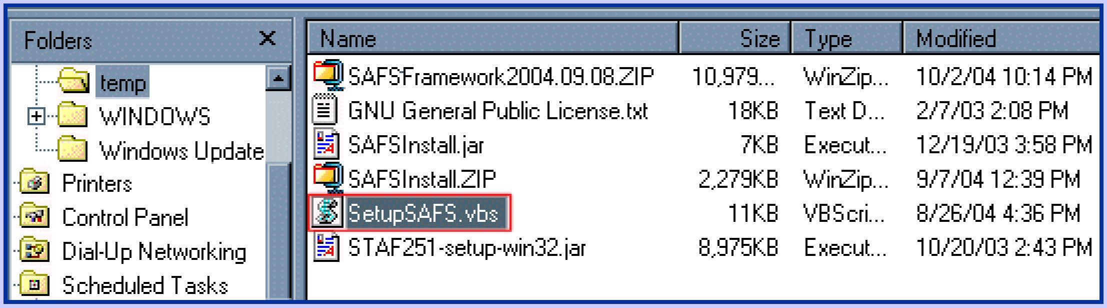


Address C:\safs\Project

| Name | Size | Type | Modified |
|---|---|---|---|
| Datapool | | File Folder | 10/3/04 12:42 AM |
| runTIDTest.bat | 1KB | MS-DOS B... | 10/4/04 9:32 PM |
| safstid.ini | 1KB | Configurati... | 10/4/04 9:18 PM |
| tidtest.ini | 1KB | Configurati... | 10/4/04 9:21 PM |

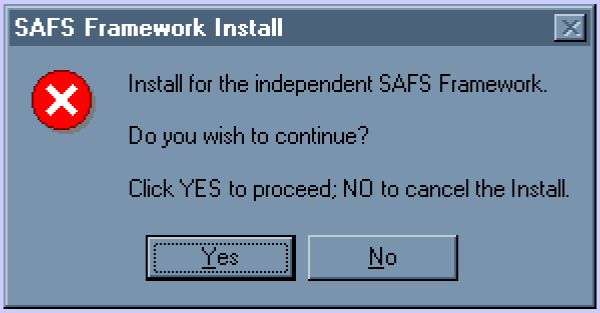
Modify the path information in all 3 of the above files to correctly point to your custom install directories. Once done with this you are good to proceed with the lesson!

## Execute the SAFS Install Script: SetupSAFS.vbs



| Name | Size | Type | Modified |
|---|---|---|---|
| SAFSFramework2004.09.08.ZIP | 10,979... | WinZip... | 10/2/04 10:14 PM |
| GNU General Public License.txt | 18KB | Text D... | 2/7/03 2:08 PM |
| SAFSInstall.jar | 7KB | Execut... | 12/19/03 3:58 PM |
| SAFSInstall.ZIP | 2,279KB | WinZip... | 9/7/04 12:39 PM |
| SetupSAFS.vbs | 11KB | VBScri... | 8/26/04 4:36 PM |
| STAF251-setup-win32.jar | 8,975KB | Execut... | 10/20/03 2:43 PM |

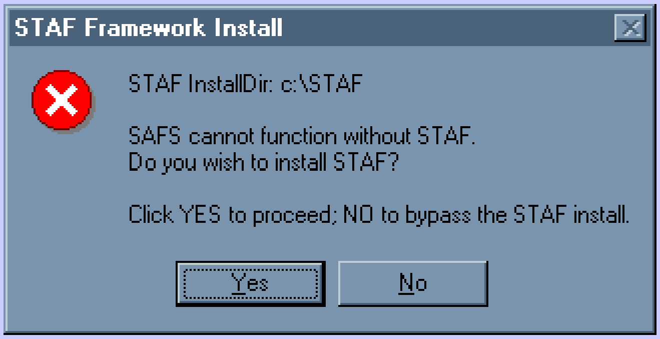
The Standard Install is not "pretty". But it does pretty much automate some otherwise complex tasks for the user. Simply **double-click** the **SetupSAFS.vbs** script to launch the installer.



Install for the independent SAFS Framework.

Do you wish to continue?

Click YES to proceed; NO to cancel the Install.



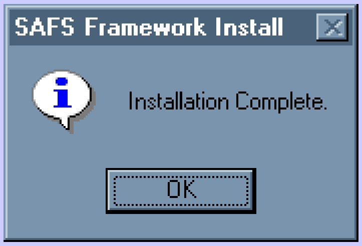
STAF InstallDir: c:\STAF

SAFS cannot function without STAF.
Do you wish to install STAF?

Click YES to proceed; NO to bypass the STAF install.

The above installer dialogs require a user response. Other dialogs may appear during the install, but they simply provide status information. They will display for a few seconds and the installation will continue.

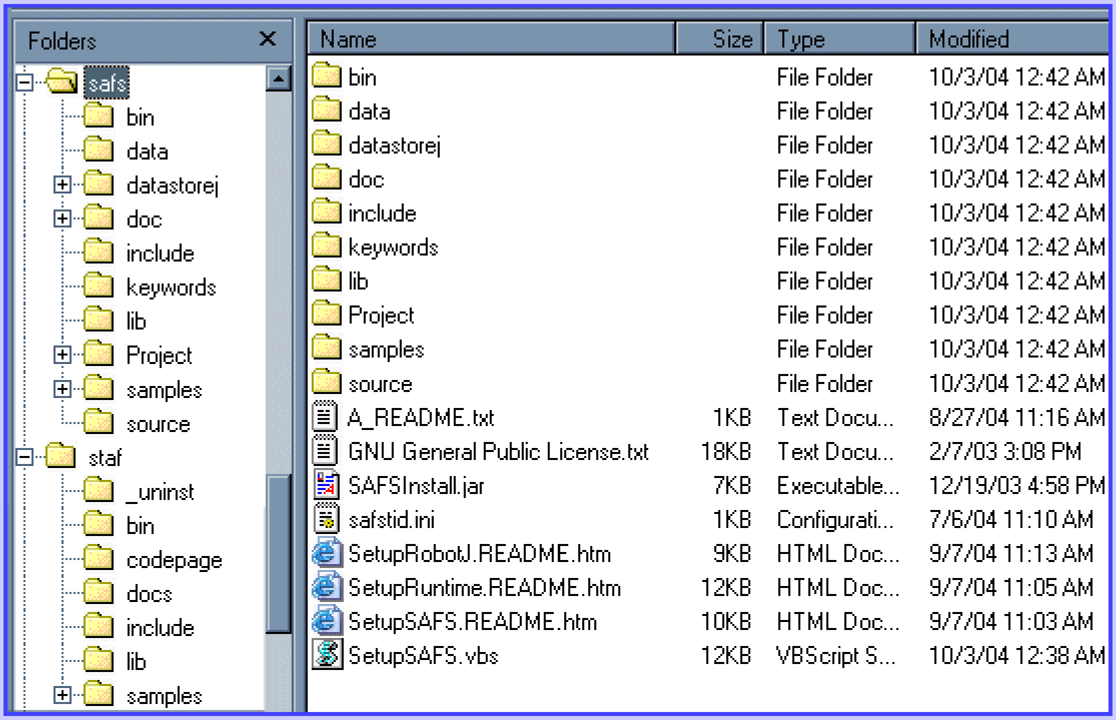
After a silent Java install of STAF lasting a minute or so, the installation will display the completion dialog shown below.



Installation Complete.

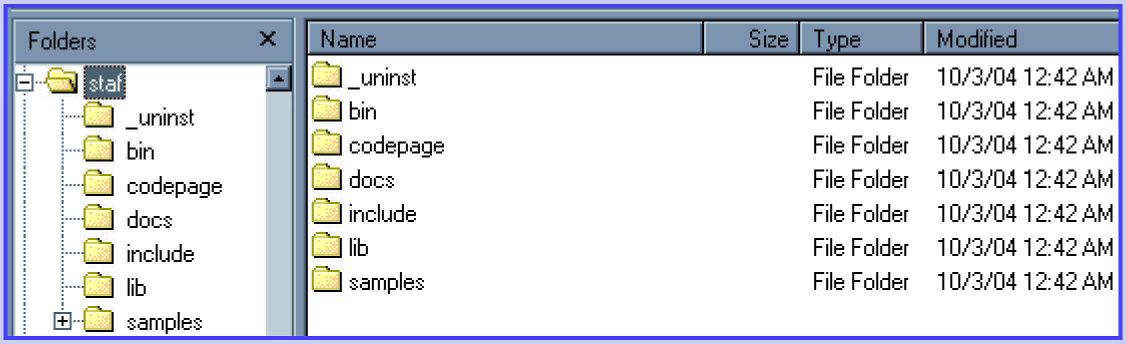
## Verify a Successful SAFS Install

Following a successful SAFS install, you should see the files and directories below in the root directory where SAFS was installed. (By default this will be C:\SAFS.)



| Name | Size | Type | Modified |
|---|---|---|---|
| bin | | File Folder | 10/3/04 12:42 AM |
| data | | File Folder | 10/3/04 12:42 AM |
| datastorej | | File Folder | 10/3/04 12:42 AM |
| doc | | File Folder | 10/3/04 12:42 AM |
| include | | File Folder | 10/3/04 12:42 AM |
| keywords | | File Folder | 10/3/04 12:42 AM |
| lib | | File Folder | 10/3/04 12:42 AM |
| Project | | File Folder | 10/3/04 12:42 AM |
| samples | | File Folder | 10/3/04 12:42 AM |
| source | | File Folder | 10/3/04 12:42 AM |
| A_README.txt | 1KB | Text Docu... | 8/27/04 11:16 AM |
| GNU General Public License.txt | 18KB | Text Docu... | 2/7/03 3:08 PM |
| SAFSInstall.jar | 7KB | Executable... | 12/19/03 4:58 PM |
| safstid.ini | 1KB | Configurati... | 7/6/04 11:10 AM |
| SetupRobotJ.README.htm | 9KB | HTML Doc... | 9/7/04 11:13 AM |
| SetupRuntime.README.htm | 12KB | HTML Doc... | 9/7/04 11:05 AM |
| SetupSAFS.README.htm | 10KB | HTML Doc... | 9/7/04 11:03 AM |
| SetupSAFS.vbs | 12KB | VBScript S... | 10/3/04 12:38 AM |

If the STAF install was not bypassed, you should see the directories below in the root directory where STAF was installed. (By default this will be C:\STAF.)



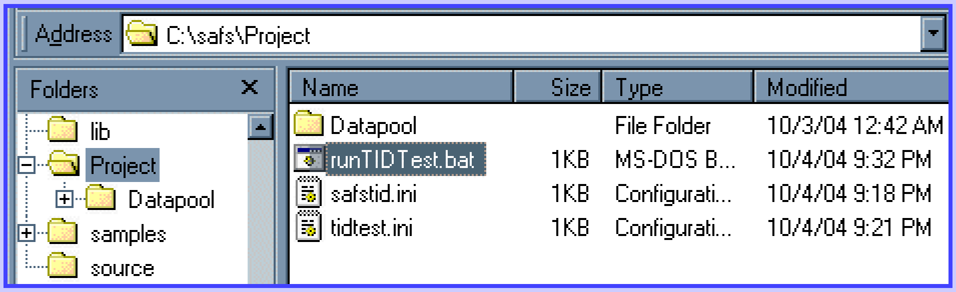
| Name | Size | Type | Modified |
|---|---|---|---|
| _uninst | | File Folder | 10/3/04 12:42 AM |
| bin | | File Folder | 10/3/04 12:42 AM |
| codepage | | File Folder | 10/3/04 12:42 AM |
| docs | | File Folder | 10/3/04 12:42 AM |
| include | | File Folder | 10/3/04 12:42 AM |
| lib | | File Folder | 10/3/04 12:42 AM |
| samples | | File Folder | 10/3/04 12:42 AM |

## Execute a Simple SAFS Test

You can also verify the installation of all components by executing a simple test installed with the SAFS Framework.

The "runTIDTest.bat" file shown below is preset to execute if you installed SAFS in the default C:\SAFS directory. If you installed to custom directories you will need to edit the path information in "runtTIDTest.BAT", "TIDTest.INI", and "SAFSTID.INI".

To execute the test, double-click "runTIDTest.bat"

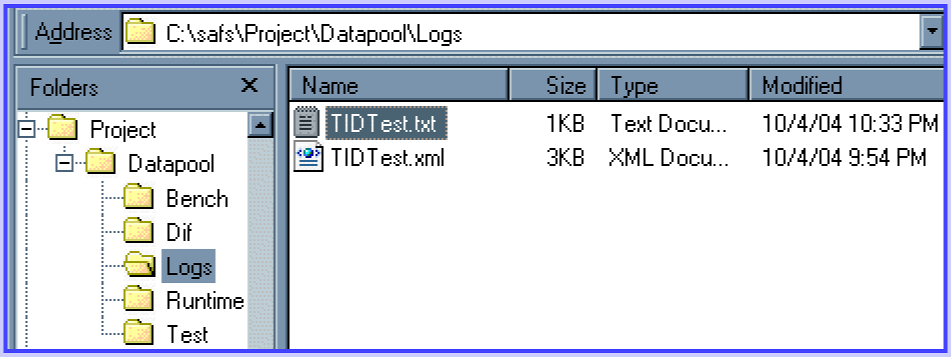


Address C:\safs\Project

| Name | Size | Type | Modified |
|---|---|---|---|
| Datapool | | File Folder | 10/3/04 12:42 AM |
| runTIDTest.bat | 1KB | MS-DOS B... | 10/4/04 9:32 PM |
| safstid.ini | 1KB | Configurati... | 10/4/04 9:18 PM |
| tidtest.ini | 1KB | Configurati... | 10/4/04 9:21 PM |

With everything working as planned, you should see a CMD window appear and the test execute. The CMD window will display information similar to what you can view here.

Upon test completion, you should have a Text log and an XML log in the Project\Datapool\Logs directory as shown below:



Address C:\safs\Project\Datapool\Logs

| Name | Size | Type | Modified |
|---|---|---|---|
| TIDTest.txt | 1KB | Text Docu... | 10/4/04 10:33 PM |
| TIDTest.xml | 3KB | XML Docu... | 10/4/04 9:54 PM |

Open that TIDTest.TXT log file and you should find a test log very similar to the one you can view here.

### CMD Window Output

**http://safsworks.com/online/image/tidout.txt**

### TIDTest.TXT Log

**http://safsworks.com/online/image/TIDTest.txt**

## Journal: Installation Graduation

The course instructors would love to read your review of this installation lesson. They can also provide some assistance if your install did not go well. You can provide such feedback in this private journal exchange with the instructors, or thru the public course forum: SAFS101 Discussion Forum.

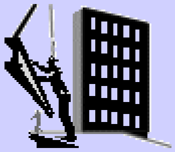
**SAFS101 Discussion Forum**

**http://safsworks.com/online/mod/forum/view.php?id=4**

**Lesson**

**3** Introducing the SAFS Framework

A brief overview of the SAFS Framework and the major tools available to any automation tool that wishes to use them.

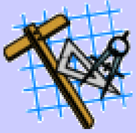


- The SAFS Framework
- Tools of the SAFS Framework
- The SAFS Framework Home Page

## The SAFS Framework

### Minimize the Impact of Change

Fig 1 shows a simplified view of the concept: "minimize the impact caused by changes in the applications we are testing, and changes in the tools we use to test them."



SAFS allows us to develop our test assets independent of the test tools we will use to execute them.  We then make use of whatever tools are available and necessary to accomplish our testing in the desired environment.

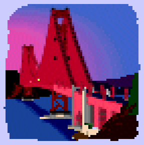
In fact, the SAFS Framework attempts to implement as much of the "ideal" automation framework discussed in our other tutorial, "Introduction to Test Automation Techniques".

So now lets take a closer look at some of the tool specifics.

**Fig 1**: http://safsworks.com/online/safs101/safsintro.gif

### SAFS Framework Tools

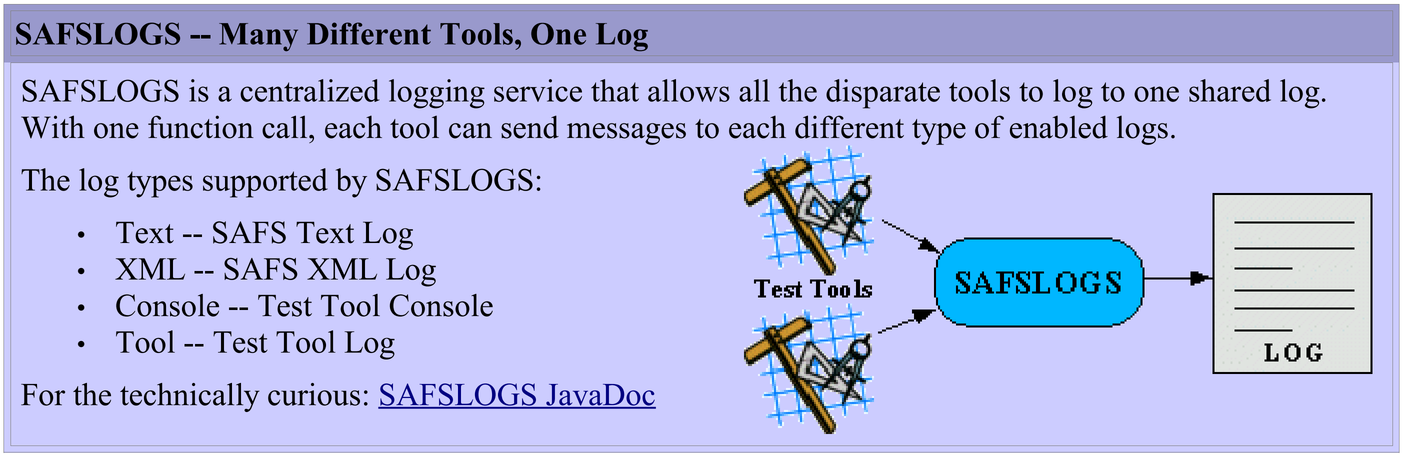
Fig 2 shows a more detailed view of tools provided by the SAFS Framework.  Included in this view are current and future SAFS engines and tools.



The framework has been developed to maximize reuse across all testing tools in all environments.  Robot, RobotJ, WinRunner, QuickTest Pro, and virtually any other tool can all bridge and connect to the same services for logging (SAFSLOGS), variables (SAFSVARS), and mapping (SAFSMAPS), etc...
This makes the framework built around each testing tool virtually identical -- not just nearly identical or "close".

Lets start examining the individual tools by taking a look at SAFSLOGS.

**Fig 2:** http://safsworks.com/online/safs101/safs.gif



## SAFSLOGS -- Many Different Tools, One Log

SAFSLOGS is a centralized logging service that allows all the disparate tools to log to one shared log. With one function call, each tool can send messages to each different type of enabled logs.

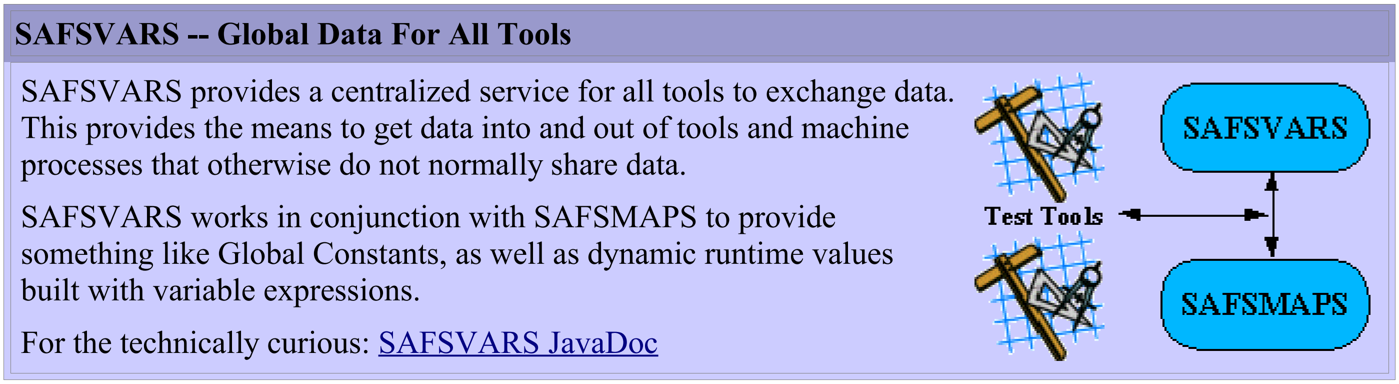
The log types supported by SAFSLOGS:

- Text -- SAFS Text Log
- XML -- SAFS XML Log
- Console -- Test Tool Console
- Tool -- Test Tool Log

For the technically curious: SAFSLOGS JavaDoc

**SAFSLOGS JavaDoc**

**http://safsdev.sourceforge.net/doc/org/safs/staf/service/logging/SAFSLoggingService.html**



## SAFSVARS -- Global Data For All Tools

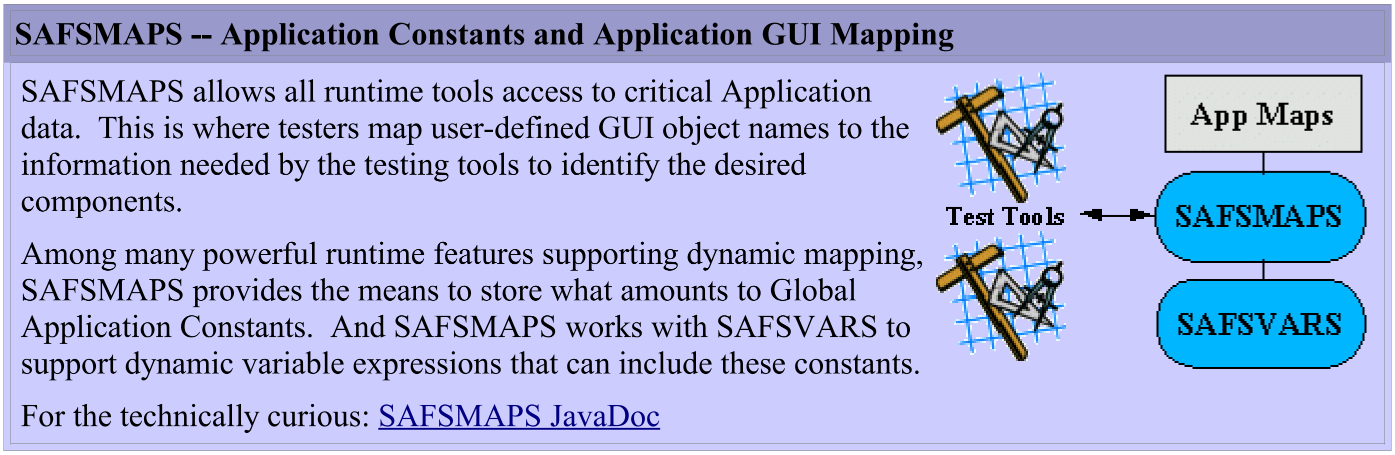
SAFSVARS provides a centralized service for all tools to exchange data. This provides the means to get data into and out of tools and machine processes that otherwise do not normally share data.

SAFSVARS works in conjunction with SAFSMAPS to provide something like Global Constants, as well as dynamic runtime values built with variable expressions.

For the technically curious: SAFSVARS JavaDoc

**SAFSVARS JavaDoc**

**http://safsdev.sourceforge.net/doc/org/safs/staf/service/SAFSVariableService.html**



**SAFSMAPS -- Application Constants and Application GUI Mapping**

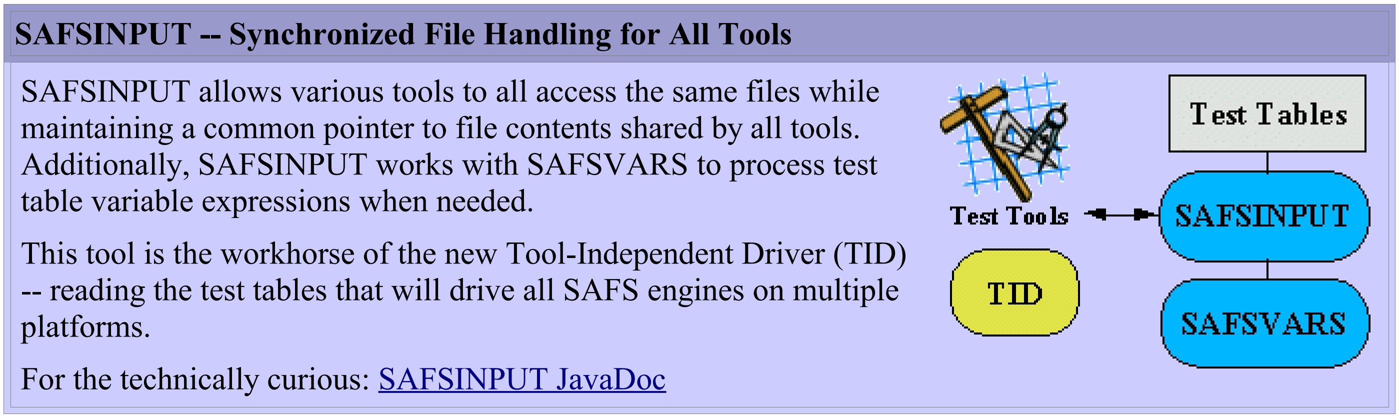
SAFSMAPS allows all runtime tools access to critical Application data.  This is where testers map user-defined GUI object names to the information needed by the testing tools to identify the desired components.

Among many powerful runtime features supporting dynamic mapping, SAFSMAPS provides the means to store what amounts to Global Application Constants.  And SAFSMAPS works with SAFSVARS to support dynamic variable expressions that can include these constants.

For the technically curious: SAFSMAPS JavaDoc

**SAFSMAPS JavaDoc**

**http://safsdev.sourceforge.net/doc/org/safs/staf/service/SAFSAppMapService.html**



**SAFSINPUT -- Synchronized File Handling for All Tools**

SAFSINPUT allows various tools to all access the same files while maintaining a common pointer to file contents shared by all tools.  Additionally, SAFSINPUT works with SAFSVARS to process test table variable expressions when needed.

This tool is the workhorse of the new Tool-Independent Driver (TID) -- reading the test tables that will drive all SAFS engines on multiple platforms.

For the technically curious: SAFSINPUT JavaDoc

**SAFSINPUT JavaDoc**

**http://safsdev.sourceforge.net/doc/org/safs/staf/service/SAFSInputService.html**

### The Technology Behind the Tools

The fact that many different automation tools can be made to exchange data is no simple task. We are talking about connecting different tools from different vendors in ways they were never designed to be connected! The automation tools are running independently, and in separate processes on the host machine.

So how does SAFS manage to get them to talk to each other?

The short answer is: STAF -- Software Testing Automation Framework.

SAFS leverages the separate open source STAF framework for all its inter-process communications. In the future SAFS will be exploiting the distributed testing capabilities of STAF, too!

For the technically curious: STAF on SourceForge

**STAF on SourceForge**

**http://staf.sourceforge.net**

## Quiz: Tools of the SAFS Framework

Were you sleeping?  I know.  Me too.  I think the server has been dozing a bit, too!

Let's see if anything from this lesson took root...

**1**

1 Marks

When evaluating expressions, sometimes the framework must turn to a service that stores application-specific, or test-specific constants.  Are these application constants retrieved from the SAFSINPUT service?

Answer: ○True [✓ ○False]

**2**

1 Marks

SAFSMAPS is used to store user-defined GUI object names and references.  In the App Maps processed by SAFSMAPS, these names are mapped to information needed by the GUI testing tools to locate and identify application components.

Does SAFSMAPS also provide storage for global Application Constants?

Answer: [✓ ○True] ○False

**3**

1 Marks

SAFSLOGS allows multiple tools to write to shared logs.  It supports several different log formats.  Does SAFSLOGS support an HTML output option?

Answer: ○True [✓ ○False]

**4**

1 Marks

SAFSINPUT sometimes must evaluate expressions in order to locate a place in a file.  What service does SAFSINPUT call upon to resolve these expressions?

Answer:
○ a. SAFS/DriverCommands
[✓ ○ b. SAFSVARS]
○ c. SAFSMAPS

**5**

1 Marks

SAFSLOGS can write to multiple log formats with a single request to log a message.  Which of the following log formats are supported by SAFSLOGS?

Answer:
□a. HTML Log
[✓ □b. Text Log]
[✓ □c. GUI Test Tool Console]
[✓ □d. XML Log]
[✓ □e. GUI Test Tool Log]

**The SAFS Framework Homepage**

**http://safsdev.sourceforge.net**

This Page Intentionally Blank

**Lesson**

**4** Test Tables

The SAFS Framework defines a new format for expressing tests. This format is independent of any test tool. Tests written in the SAFS format are portable to any number of tools and platforms.

Intro to SAFS Test Tables

SAFS Test Table Specifications

## Intro To SAFS Test Tables

### Test Tables vs. Scripts

In order for SAFS to provide true tool independence, a new format for test "scripts" was devised.  The goal was to provide an easy to read format suitable for both humans and software that could migrate to multiple platforms.

SAFS developers chose simple text files with each line representing a single instruction or action. The keywords and parameters that make up the instruction are separated by user-defined delimiters -- usually a TAB or a COMMA.  So it is ,essentially, a simple table format with delimited fields.

Below is an example of what a basic Step Table might look like:
(more on what is a Step Table on the next page)

| | | | | |
|---|---|---|---|---|
| C | SetApplicationMap | MainAppMap.map | | |
| T | MainWindow | UserID | SetTextValue | ^userid |
| T | MainWindow | Password | SetTextValue | ^password |
| T | MainWindow | Login | Click | |

This format allows test designers to use their favorite text editor, a spreadsheet program like MS Excel, or even a true database with special views and entry forms.

### Do SAFS Engines process tests from MS Excel or ODBC Databases?

SAFS test tables are simple text files with delimited fields. This allows test designers to use spreadsheet programs like MS Excel, or more formal database programs with views and forms for editing.

So does that mean that SAFS Engines read tests from MS Excel or databases?

○ Yes. SAFS Engines can process tests from any spreadsheet program or database.

○ No. SAFS Engines do not read from these directly.

## 3 Levels of Test Table Definitions

In SAFS there are 3 types or levels of test tables: Step, Suite, and Cycle.

You have already seen a sample Step Table. Step Tables contain a group of simple actions like setting a text value (SetTextValue) or clicking a button (Click). These types of actions -- called Component Functions -- can only be done in Step Tables, and only a few of these simple actions are used in each Step Table to perform a minor task.

For example, there would be separate Step Tables to:

- LaunchMyApplication
- Login
- CloseMainWindow

Suite Tables are used to form test suites -- more complex operations formed by calling various Step Tables in a desired sequence.

Below is a trivial Suite Table calling the 3 Step Tables mentioned above:

| | | | |
|---|---|---|---|
| T | LaunchMyApplication | | |
| T | Login | ^userid=myuserid | ^password=mypassword |
| C | Pause | 15 | |
| T | CloseMainWindow | | |

Finally, a Cycle Table is yet a higher level formed by calling various Suites in a similar fashion.

## What Type of Test Table Contains Simple Component Functions?

We mentioned that there were 3 levels of test tables in SAFS: Cycles, Suites, and Steps. These actually form a hierarchy of test tables that help group simple actions into complex activities and then larger test processes.

Component Functions are those simple actions used to act on the application components. Which type of test table is used to perform these simple actions?

○ [✓ Step Tables]
○ Cycle Tables
○ Suite Tables

## The Separation of Roles through Test Tables

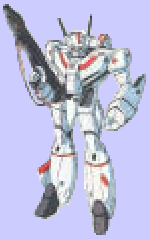
SAFS users often talk about "High-Level" test tables and "Low-Level" test tables.  This differentiation is intended to separate the testing role responsibilities where applicable.

Test Designers generally are responsible for the High-Level test tables -- the Cycle tables and Suite tables.  Test Automators are generally responsible for fleshing out the Low-level test tables.

If you review the Classics example in the SAFS Install directory . *\SAFS\samples\Classics\ClassicsC_V2001.zip* you would find that there are two separate Excel Workbooks:

- ClassicC_High.xls
- ClassicC_Steps.xls

These serve as excellent (alright, maybe just 'OK') examples of the difference between the High-Level test tables developed by the Designer and those of the Automator.  The biggest thing to notice is that the Designer's High-Level test tables generally tell us what needs to be done, but do not specify the nitty-gritty steps of how each will be done.  The Automator's Low-Level step tables will flesh out those nitty-gritty details.



**SAFS Test Table Specifications:**

**http://sourceforge.net/docman/display_doc.php?docid=17266&group_id=56751**

This Page Intentionally Blank

**Lesson**

**5** Application Maps

Map tester-defined component names to the information needed by the test tools to locate the components on the screen. We can also provide data "constants" or stored values that can be referenced in test tables or used in variable expressions.

- Not Your Typical GUI Object Map
- Dynamic Component Recognition Support
- Evil Developers Wreak Havoc on Test Automation!

## Not Your Typical Object GUI Map

### What is an App Map, or GUI Object Map?

Most automations tools provide something along the lines of what SAFS calls an Application Map. These may be called GUI Maps, Object Maps, or many other things; but they generally all satisfy a similar need.

These Maps allow test scripts to reference application components or objects with simple names. They provide a means to "map" those simple names to the more sophisticated and detailed information needed by the automation tool to locate and manipulate the desired object.

Example of a fictitious tool map:

```
MainWindow:
    Class=com.safsworks.JFrameClass;
    Domain=Java;
    Caption=Welcome To SAFSWorks;
    ObjectClass=javax.swing.JFrame;
    ObjectType=Window;
```

Some tools use more, or less, information to accurately identify an object. The important thing is, the scripts only need to reference the component "MainWindow", and the Application Map will fill in the details for the automation tool.

## Why Use an Application Map?

Suppose for a minute that you are writing a document for someone who knows nothing about your application, and you need to specify in that document an accurate description of each component. Keep in mind the reader doesn't know which application is being discussed, and doesn't know which application is being tested among all the applications that might be running on the machine.

Well, that is the environment of the automation tool, and it needs very specific information so it can locate the right component in the right application. Think of the Application Map as a sort of "Glossary" or "Dictionary" for the automation tool.

Our previous MainWindow example listed unique properties for the MainWindow component. It would be terribly cumbersome to place all that information in our tests every time we wanted to touch the MainWindow.

```
Caption=Welcome To SAFSWorks;
ObjectClass=javax.swing.JFrame;
ObjectType=Window;
```

Actually, it would be automation suicide !

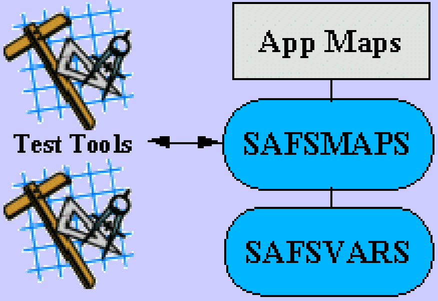
If tests duplicated all that information everywhere, what would happen to those tests when the Caption of the Window changed? The answer: Every one of those tests would now be unusable! We would have to find and fix every test that used the old Caption.

The Application Map allows us to place all that fragile information in a single place that is easy to maintain. When the Caption does change, the tests will still stop working. But we don't have to fix the tests because these only reference "MainWindow". We fix the Caption in the mapped definition of "MainWindow" in the Application Map and all the tests will work again.

## Features of SAFS App Maps

SAFS App Map support goes beyond what many tools provide. The SAFSMAPS service works in conjunction with SAFSVARS global variable storage to provide these interesting features:

- Unique Object Mapping per Window
    - Object is unique to this Window
- Shared Object Mapping for multiple Windows
    - Object is the same in multiple Windows
- Application Constants storage for variable reference
- Dynamic Object Mapping with variables and constants



Together these features provide very powerful means to identify components statically, or dynamically at runtime. More importantly, it reduces maintenance incidents even more than would be typical using other Application Mapping systems.

## Format of a SAFS Application Map

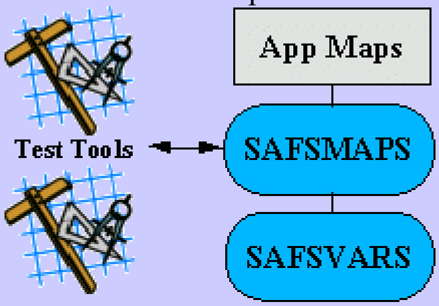
Like Test Tables, SAFS App Maps are simple text files. This makes them easy to read, edit, and migrate to multiple platforms. Entries in the App Map are Name=Value pairs divided into sections that represent windows or containers. An exception to that is a default section named "[ApplicationConstants]" which can contain data other than component information.

A simple App Map example:

| | |
|---|---|
| [ApplicationConstants] | <- Default Constants Section |
| DefaultUserID=defaultUser | <- A Constant |
| DefaultPassword=defaultPassword | <- A Constant |
| ;OK button same everywhere | <- A Comment |
| OK="Type=Pushbutton;Text=OK" | <-A Shared Component |
| [Google] | <-A Window Section/Name |
| Google="Type=Window;Caption={Google*}" | <- A Window Definition |
| Logo="Type=HTMLImage;HTMLID=Google" | <- A Child Definition |
| Search="Type=Pushbutton;Text=Google Search" | <- A Child Definition |

## SAFS Shared App Maps

While SAFS test tables are not "test scripts" in the traditional sense of a tool-specific scripting language; in the deepest, darkest bowels of a typical SAFS Framework install there is usually a GUI automation tool that needs component information. Thus, we use SAFS App Maps to satisfy this craving.



Finally, to mention in passing: An exciting topic for some future tutorial is how SAFS is working on a common object definition syntax and algorithm intended to be used with all SAFS Engines -- regardless of the underlying automation tool. This means SAFS users would be able to use a single App Map that will work for all SAFS engines and testing tools. This is not so farfetched. It is already working for IBM Rational Robot and XDE Tester!

The alternative requires a different App Map for each supported GUI testing tool -- something ease-of-use dictates the SAFSDEV engine developers circumvent.

**Dynamic Component Lookup Support:**

**http://safsdev.sourceforge.net/sqabasic2000/CreateAppMap.htm#ddv_lookup**

## Evil Developers Wreak Havoc on Test Automation

I only partially disagree with that little intuitive voice inside me that says, "Developers *are* Evil." 😈

We had been automating a rather poorly designed web application for a few weeks when things went abruptly bad. Our automation could no longer enter the application. We could no longer log in. 😕

We quickly isolated the problem. The problem was: logging in to the application was no longer valid. We now needed to log "on"! 😲 Yes, "Login" was history. "Log On" was the text of the day. Our tests could not log onto the application because the link and button text had changed!

Fortunately, we did not have to hunt down a gazillion tests to fix this. We made the change from "Login" to "Log On" in the Application Map on one line and everything worked again. 😄
That is, until we went through a similar issue a few days later when "Logoff" became "Log Off". Doh! 😠

And yet again when they so *nicely* switched it back to "Logoff" a few days later! ☹

OK, maybe all developers aren't really "evil". Maybe they're just "playful". Yeah, maybe that's it. 😉

**Lesson**

**6** DriverCommands and ComponentFunctions

The two main types of predefined keywords or actions available for test tables. Having these low-level actions available means there may be little or nothing the test automator needs to code. Most common test activities are already written for you!

Driver Commands and Component Functions

Keywords Quick Reference

Keywords Detailed Reference

## Driver Commands and Component Functions

### What are Driver Commands?

As the name suggests, Driver Commands are generally commands that instruct the test Driver to perform some task. These usually don't involve the application GUI. Instead, Driver Commands are for exciting things like logging user-defined messages, launching applications, incrementing status information, and performing flow control tasks. In some cases, the commands are actually sent to running engines to affect some change in engine behavior.

The Keyword Reference has a separate section for Driver Commands, and these commands themselves are shown in separate categories like DriverFlowCommands, DriverFileCommands, and DriverStringCommands, among others.

**In the Keyword Reference linked above, which engines are listed as supporting the DDDriverCommands keyword "*Pause*"?**

□ RJ
□ RC
□ SDC
□ WR

**Keyword Reference:**

**http://safsdev.sourceforge.net/sqabasic2000/RRAFSReference.htm**

**Driver Commands:**

**http://safsdev.sourceforge.net/sqabasic2000/DriverCommandsList.htm**

## What are Component Functions?

Component Functions are the keywords that act on GUI components. They require an underlying GUI automation tool that can examine and manipulate the controls of the application.

The Keyword Reference has a separate section for Component Functions. These are broken down by the component type supported, and each subsection lists the keywords supported for that component type.

**In the Keyword Reference linked above, which engines are listed as supporting the TreeViewFunctions "*SelectTextNode*" keyword?**

☐ SDC
☐ RJ
☐ RC
☐ WR

**Keyword Reference:**

**http://safsdev.sourceforge.net/sqabasic2000/RRAFSReference.htm**

**Component Functions:**

**http://safsdev.sourceforge.net/sqabasic2000/ComponentFunctionsList.htm**

**Keyword Quick Reference:**

**http://safsdev.sourceforge.net/sqabasic2000/RRAFSQuickReference.htm**

**Lesson**

**7**

## Current SAFS Engines and the Future

From Rational Robot to Mercury WinRunner and more, see what exists now and what is coming soon!

- Current SAFS Engines and Future Direction
- SAFS Overview Graphic
- Current Development on SourceForge

## Current SAFS Engines and Future Direction

### RRAFS -- SAFS Engine for IBM Rational Robot

RRAFS stands for Rational Robot Automation Framework Support. RRAFS was the very first SAFS engine, has the most features, the greatest stability, and serves as the standard all other SAFS engines emulate whenever possible. The keyword reference doc annotation for this engine is "RC".

RRAFS is a standalone implementation. This means it is entirely self-contained; with a fully functional "Driver" and "Engine". Consequently, it was designed to be used this way. It cannot yet be used in an engine-only mode--a mode that would allow the upcoming Tool-Independent Driver to call RRAFS. That will surely come in time, however.

As a fully functional STAF-aware Driver, an important thing RRAFS can do is call other running engines. RRAFS can currently call the SAFS/DriverCommands, and SAFS/RobotJ engines. When new engines become available, RRAFS can easily be updated to take advantage of those, too!

Question:
**Is Rational Robot able to take advantage of the latest features in the Java-based SAFS Framework?**

□ No, Robot doesn't know Java
□ Yes it can through STAF

## WRAFS -- SAFS Engine for Mercury Interactive WinRunner

WRAFS stands for WinRunner Automation Framework Support. WRAFS was the second SAFS engine and was developed a few years after the first RRAFS engine. It was a translation of a snapshot of RRAFS code performed by John Crunk. The keyword reference doc annotation for this engine is "WR".

WRAFS is a standalone implementation. This means it is entirely self-contained with a fully functional "Driver" and "Engine". Consequently, it was designed to be used this way. It cannot yet be used in an engine-only mode -- a mode that would allow the upcoming Tool-Independent Driver to call WRAFS. That may come in time, however.

WRAFS is not quite as full-featured as RRAFS, but it is beginning to use other SAFS engines like SAFS/DriverCommands. The latest WRAFS code is STAF-aware and users can begin to take advantage of the common SAFS Framework. There are actually several developers actively working to enhance WRAFS to its fullest potential!

**So, does WRAFS do everything that RRAFS can do?**

□ I think so

□ Not quite

## SAFS/RobotJ -- SAFS Engine for IBM Rational Functional Tester

SAFS/RobotJ is the first fully STAF-enabled engine. It is a "pure" engine in that it has no Driver, and MUST receive commands from an external Driver through STAF. It cannot run standalone like RRAFS and WRAFS. The keyword reference doc annotation for this engine is "RJ".

Until the SAFS TID (Tool-Independent Driver) is completed, the only Driver really setup to use SAFS/RobotJ is RRAFS. In that scenario, testers get to use *both* IBM Rational Robot *and* IBM Functional Tester ***at the same time***. They don't have to pick one tool over the other. They get the best of both! (Assuming, of course, they have both tools--which many do.)

SAFS/RobotJ is also the first engine that has fully implemented support for the SAFS GUIVector Algorithm (SGVA). This allows IBM Rational Robot and Functional Tester to share the same SAFS AppMap and locate components from a standard set of recognition strings. Thus, Functional Tester is able to locate components with Robot recognition strings!

**Can SAFS/RobotJ call other STAF-enabled engines?**

□ No it can't

□ Sure, it's the best

### SAFS/DriverCommands -- SAFS Engine for Shared Driver Commands

SAFS/DriverCommands is the first completely vendor-independent SAFS engine! It does not require any fee-based tool. It does not need IBM Rational Robot, Functional Tester, or WinRunner to function. The keyword reference doc annotation for this engine is "SDC".

Like SAFS/RobotJ, the SAFS/DriverCommands engine is a "pure" STAF-enabled engine. It has no Driver, and cannot run standalone like RRAFS and WRAFS. It requires an external Driver to send it commands to execute.

Until the SAFS TID (Tool-Independent Driver) is complete, the only Driver really setup to use SAFS/DriverCommands is RRAFS. The SAFS TID actually does use this engine, but the TID itself has not yet been released for production.

This engine is intended to only provide Driver Commands. As such, it doesn't require a GUI testing tool, though one may be added at a later date if needed.

**Do I have to have to buy a commercial tool to use the SAFS/DriverCommands engine?**

□ Yes, only IBM Rational Robot can use this engine

□ No, it is completely free

### SAFS/JRex -- A Multi-Platform SAFS Engine for Web Testing

SAFS/JRex is the first "pure" SAFS Engine that will provide full Web GUI testing capabilities on multiple platforms. The browser used for client testing will be the Java-based JRex engine currently in Open Source development at JRex on MozDev. The keyword reference label for SAFS/JRex will be "SJR".

SAFS/JRex is still in the intermediate stages of development by Windows and Linux teams at this time. The SAFS TID will be the Driver of choice for this engine when testing on non-Windows platforms, but RRAFS will be able to take advantage of SAFS/JRex, too, once it is ready for production.

**Will SAFS/JRex support testing with Internet Explorer or Netscape?**

□ Not likely

□ Of course

**JREX on MOZDEV:**

**http://jrex.mozdev.org/**

## SAFSDRIVER -- A SAFS Driver for Multi-Platform Execution

When the SAFS community talks about the "Tool-Independent Driver", or "TID"; they are talking about SAFSDRIVER. This is the first vendor-independent Driver in the SAFS arsenal. It is intended to provide multi-platform Driver functionality that requires no fee-based tool for execution.

The TID is in the *late* stages of development. It can use the SAFS/DriverCommands, SAFS/RobotJ, and the SAFS/JRex engines. But, the TID does not yet support a number of the Driver Commands and core features that a full-featured Driver like RRAFS must support. Developers are actively working on the TID at this time.

**Can SAFSDRIVER be used to run tests with RRAFS or WRAFS?**

□ Yes, but only on Windows

[✓ □ Not at this time]

## Ongoing Development and the Future

Currently, SAFS developers are hard at work enhancing existing engines, creating new engines, and working on adding more multi-platform capabilities.

Ongoing development:

(all this is happening in parallel!)

- Enhancements to RRAFS and WRAFS
- Enhancements to SAFS/RobotJ
- Multi-Platform enhancements to SAFS/DriverCommands Engines
- Multi-Platform SAFS/JRex Web Engine
- Multi-Platform SAFSDRIVER Tool-Independent Driver

Future Plans and Possibilities:

(not in any particular order)

- Multi-Platform support for SAFS/RobotJ
- Engine for Web Testing with Internet Explorer and Netscape Navigator
- Engines for Abbot, JFCUnit, Marathon, or others per demand
- Refactor RRAFS and WRAFS to accept SAFSDRIVER control
- Explicit support for distributed testing and virtual users
- API Testing (Unit/System)

**Would you agree that SAFS is a thriving project with good long-term goals?**

□ No. I just don't get it.

[✓ □ Absolutely!]

□ Maybe. I haven't followed it long enough to know.

**SAFS Framework Overview Graphics:**
**http://safsdev.sourceforge.net/doc/SAFSFramework.htm**

**SAFS Java Development Implementation Graphics:**
**http://safsdev.sourceforge.net/doc/JSAFSFrameworkGraphs.htm**