

savvytest

better software
in less time

Technical Product White Paper



Königsallee 43
71638 Ludwigsburg
Germany

www.savignano.net/savvytest

Abstract

savvytest is a software tool which enables you to perform tests on your software components in an easy and professional way. No matter if you need to test your own new software, changes and enhancements, refactoring, or even third-party software. Since the concept is based on black box testing of interfaces, it is independent of their implementation language. savvytest supports testing of components as a base of

- ✓ module tests
- ✓ integration tests
- ✓ build tests
- ✓ regression tests
- ✓ acceptance tests

Although especially aimed at testing mainframe components, savvytest is designed to provide a single approach for all your software, independent of the target platforms.

Because savvytest separates the technical interface from the business view, both developers and business testers can provide test scenarios based on their own expert point of view. Their test scenarios can be exchanged or used together to test multiple aspects of an interface, thus allowing for more complete tests.

For non-technical users, an extra application with a simplified GUI can be provided. It allows to easily provide test scenarios without the need for any in-depth knowledge of the internal implementation of the interfaces. However, the application nevertheless works on the same data base as the developer's plugin.

The developer is provided with a user interface integrated in his or her usual developments environment, the Rational Developer for System z. This allows for fast and easy access to tests during the

development and integration cycle, without the need to get used to a completely new tool interface.

Concept

The design of savvytest is conceptually based on the following principles:

- It should be easy to perform tests, otherwise tendency is, they won't be done. This means that the tool must be simple and easy to handle.
- Both, the business and the technical view point, are necessary for complete and reliable tests. This means that there synergetic effects to be exploited.
- Even in a technically heterogeneous environment, it should not be necessary to use different tools for the same task. This means that we want to provide one tool for all.

Easy to handle

The developer's front-end is integrated in his or her development environment by providing savvytest as a plug-in to Eclipse based IDEs. For the System z target platform, the plug-in is especially designed to integrate with IBM's Rational Developer for System z.¹ It can therefore make use of its mainframe integration, allowing the user not only to define test interfaces and test scenarios, but also to run and control in the same place.

Furthermore, as savvytest makes use of the Eclipse interface concepts, the look-and-feel also integrates tightly and does not require the user to learn new concepts.

¹ Rational Developer for System z by IBM (also often referred to as RDz) is the successor to the Websphere Developer for System z (WDz), and is based on the Eclipse platform.

Achieve synergies

It is one of savvytest's important conceptual principles that a complete and reliable test requires more than just the business or the technical view, but both. Usually, test tools are designed for use by the developer or by the business tester, but not by both at the same time. The disadvantage of using different tools is that neither the business tester can benefit from the developer's test scenarios nor vice versa.

Although savvytest can provide a simplified user interface for non-technical users, delivered as an RCP client application, the different types of scenarios are not impacted by the different user interfaces. Therefore, both the developer and the business tester can mutually benefit from each other's work. Since both scenarios work on the same data, what they produce can even be easily exchanged.

In the end, integration, regression, and acceptance tests can merge all of these test scenarios and thus benefit from the various points of view they contain, broadening the test base remarkably.

One for all

Nowadays, heterogeneous architectures are widespread, and there are good reasons for this. Mainframes, Unix servers, Windows machines – all of these platforms often coexist under one roof. However, this heterogeneity does complicate a few aspects of life, and testing in one of them. Usually, the different platforms do not only require different technical skills, but they also require the use of different test tools.

savvytest is designed to provide one consistent access to testing, irrespective of the target platform or the implementation language. This means that there is only one way to define test scenarios, enabling users

to provide these for any platform or implementation language.

This is done by transforming structural and implementation specifications into a language and platform independent representation. First, structural specifications described in a specific implementation language are translated into a language independent structural description. Then, at runtime, this description is used to convert the test data into the target platform's specific binary representation. After the test target has been executed, the binary results are converted back. This means, that the specifics of the implementation language, as well as those of the target platform are strictly hidden not only from the tool's core, but also from the user specifying the test scenarios.

How to

Below, we describe a typical approach for testing a component. Since savvytest in its first release concentrates on the support of System z targets, a typical COBOL target is used to describe the steps in detail.

savvytest requires two steps in order to prepare for the definition of test scenarios:

1. Import of structural definitions
2. Definition of the interface

First of all, the structural descriptions are imported. Since structures are usually described in a programming language like COBOL, a mechanism is provided to transform these into language independent structural descriptions – short term “LiStructure”. This transformation is usually referred to as “import”. Any implementation language can be supported by providing an adequate plug-in. In fact, the language does not need to be a programming language like C, COBOL, PL/I or even Assembler, but instead can be any type of

descriptive data like an IDL file, an XMI model, or even a plain CSV list.

Once the structural descriptions have been transformed, the next step is to define interfaces by specifying the target and the required argument structures. The structural information is then augmented by specifying the usage type – input, output, or both – of each structure or part thereof. This is, strictly speaking, a non-technical specification, as the usage is defined by the functionality of the interface. We therefore add a business functional point of view on top of the technical interface.

Now, the developer as well as the business tester can start defining test scenarios based on the defined interface. Test scenarios basically consist of two types of data:

- Test input data:
data which are sent as input to the test target
- Check conditions:
data which are to be checked after the test target has been executed

In the test data editor, the user will only see relevant structures and fields therein, i.e. those defined as input or input and output data. Therefore the view at this point is very specific to the function implemented by the test target. It is more the view of the business service – a fact which helps non-technical testers to define appropriate test scenarios.

The check conditions can be provided in an easy way by defining so-called “simple check conditions”. These are check conditions that basically check if output data fields contain certain values. In some cases, these simple check conditions may be too simple to serve a required purpose. If so, you can switch to the “complex check conditions” where you can check virtually anything that can be described in the full

featured Java Expression Language (JEXL).²

Eventually, a test suite defines a set of test scenarios. When the test suite is executed, all test scenarios contained therein are sent to the target platform which in our example would be a mainframe running zOS.

The execution of a test suite consists of three steps:

1. BinaryBuilder: Based on the structural description, it converts the test data into their platform specific binary representation.
2. TargetRunner: Using these binary data as call arguments, it executes the target which in our case would be the load module of a COBOL program.
3. BinaryParser: Again based on the structural description, it converts the target's binary output back into the language independent data representation.

After the successful execution of the target (in the sense that it was technically successful), the ResultChecker reads the output data of each test scenario and applies the check conditions associated with it. The results are saved and then transformed into a customer specific format like an HTML report, a CSV file, or any other type of XML data file.

² JEXL by Apache Commons, see <http://commons.apache.org/jexl/>

Customizing

savvytest is based on two major principles:

- Language, platform, or customer specifics can be provided by its plug-in architecture
- Due to its XML based data model, it is fairly easy to interchange data with other tools at almost any point

This means that it is a basic concept of savvytest to allow for customizing to meet your special needs.

Why is savvytest just right for you?

If you are looking for a reliable and easy-to-use tool to test your own or any third-party mainframe components or services, to the best of our knowledge, savvytest is the most complete solution available.

If you are looking for a way to unify your test approach in a heterogeneous IT architecture, savvytest can help you find a consistent solution.

If you are looking for a way to start building up a regression test data base, but you do not take the risk to be bound to a specific tool, savvytest is your friend, since all of your precious data are stored in XML, allowing for easy retrieval and migration – although we do not believe that you will really want that.

Contact

If you want to know more about savvytest, you are welcome to get in touch with us.



Königsallee 43
71638 Ludwigsburg
Germany

Tel. +49-7141-890405
Fax +49-7141-890407

www.savignano.net
savvytest@savignano.net