2008

Unlock the Powers of Test Automation with Testability and Abstraction layers

Jamo Solutions NV

2/19/2008

Introduction

Today most test automation tools provide record/replay features to script test cases. However, organizations that take test automation seriously, often implement a testing framework on top of record/replay tool. The objective of the testing framework is to increase the benefits of test automation and to support all different kinds of testing and testers which is difficult to do with the current testing tools in the market.

This white paper is an introduction to a best practice in setting up a testing framework for testing software applications.

Firstly, the paper discusses the need of a complete and detailed testability interface, which is the "glue" between the automation tool and the software under test. We argue that without this kind of glue it will not be meaningful to do automation of any test cases as the benefits will be less than the effort of automation. In order to deal with complex test cycles, high volume test cases and regression test cases, the paper argues that on top of this testability interface, it is best to create additional abstraction levels in order to increase the total ROI of test automation.
Abstraction helps to:

- Isolate test cases from changes
- Enable test building for people with less technical skills by high abstraction or graphical notation
- Enable fast test case creation
- Enable reuse by sharing implementation and other assets
- Enable the whole spectrum of different testing (no strict guidelines and limits)

The reader is invited to compare his current automation testing efforts with the described best practice in order to quantify the ROI of his efforts.

Testability

The prerequisite to the automation is the completeness of the testability interface provided by the system under test.

An automation tool is built around the information that the tool can retrieve from the system under test. The system under test exposes an API that allows the automation tool to retrieve information on the status of the system under test and that allows the automation tool to drive the system under test. This API is called the testability API of the system under test. It is important that this testability API needs to efficiently deliver complete and accurate information to the automation tool.

You can compare it with following situation: The automation tool is a blind man. The system under test is a room that the blind man needs to explore. The testability API is a man inside the room who shouts to the blind man the names of the objects that are in his range so that the blind man can explore the room and deliver a description of all the objects.

If the man inside the room shouts incorrect information to the blind man, then the blind man encounters problems in describing correctly the objects that are inside the room. For example, it can happen that the testability man tells the blind man that both objects shown below are a chair.

For us, who can see it, it is clear that the left object is indeed a chair however the right object is a table. The testability man is giving incorrect information. When the blind man reaches the table, he will find out that it is not a chair by touching the object with his hands. This is an additional effort that the blind man needs to make. The effort to deal with incorrect information can also lead to errors: what can be the following item? Is it a chair or a table for children?

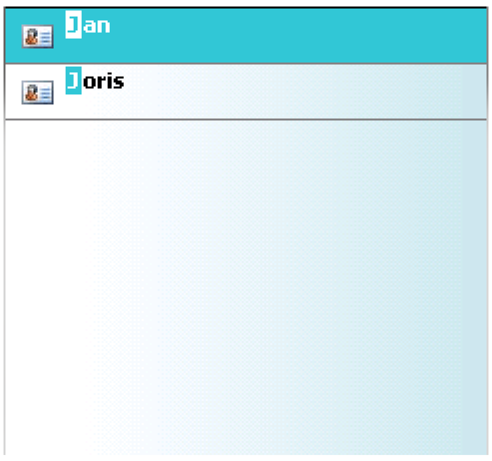
An automation tool is a program and thus by default contains no intelligence. The automation tool is like the blind man exploring the room. The testability interface of the software under test will help the automation tool in discovering the characteristics of the software under test and it is essential that this information is complete, correct and detailed enough to enable multiple levels of testing.

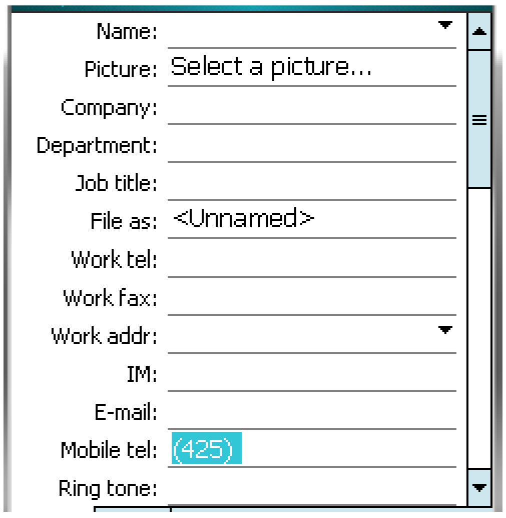
While we perform a manual test, we use all the capabilities of the human being, i.e. intelligent recognition of what is happening with the software under test. The human being is also able to “read between the lines” and spot problems that are not explicitly described to be verified in the test case.

In many cases people try to automate the same script that is meant for manual execution. This leads to incomplete testing because it implicit verification done by human beings in a manual test case is not included in the automation test case..Instead of translating a manual test case one to one in an automation test case, it is necessary to make the automated test case to look and to verify for additional information, the automated test case needs to give much more detailed instructions to the testing tool and also perform additional sanity/status checks on the software under test in order to resemble more the capabilities of the human tester.

It can be that the testability interface to the SUT does not exist. For example in mobile device domain most testing tools try to read the bitmap and try to do the verification by using OCR or shape recognition. We claim that in this case the effort of scripting the test will be bigger than making the same test manually. With this kind of “analog” verification one could also argue if they are verifying anything as the difference of a human being and bitmap verification is huge: bitmap verification is always a very small part of the screen and on the other hand, the human is able to verify the whole screen by one brief glance even if this is not explicitly mentioned in the test case. In our opinion, manual testing is more suitable to environments where there is no proper testability interface.

In Windows Mobile OS we have the capabilities of testability but the interface is not exposing detailed information all the time. For example, the testability interface indicates that following both UI components are syslistview32 UI objects.


Jan
Joris


Name:	
Picture:	Select a picture...
Company:	
Department:	
Job title:	
File as:	<Unnamed>
Work tel:	
Work fax:	
Work addr:	
IM:	
E-mail:	
Mobile tel:	(425)
Ring tone:	

The both screen have however a different semantic meaning. The left screen behaves like a normal list, the right screen acts like a sheet on which you can fill in data. The naming syslistview32 is covering several implementations and is thus not detailed enough. The tester will have to make a difference in his approach to the right and left screens. He will specialize in his testing framework the notion syslistview32 either into a normal list control or to a data record control..

The implementation and maintenance of such work-around testability frameworks reduces the re-use of the test scripts in regression testing and leads to an additional cost factor for test automation.

In general, the information delivered by the testability interface needs to be complete and detailed enough in order

- **to drive the software application automatically,**

- **to perform verification against any desired information elements,**
- **to synchronize on statuses reached by the application and to handle exceptions.**

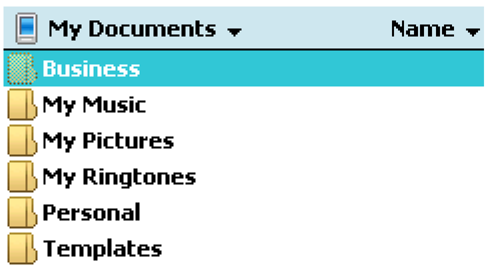
If the testability interface is incomplete or not giving detailed information, then the QA engineer will need to investigate if he can create a framework to compensate these weak spots in the testability API. If this framework is very complex and/or very difficult to maintain, one should need to consider manual testing instead of automation.

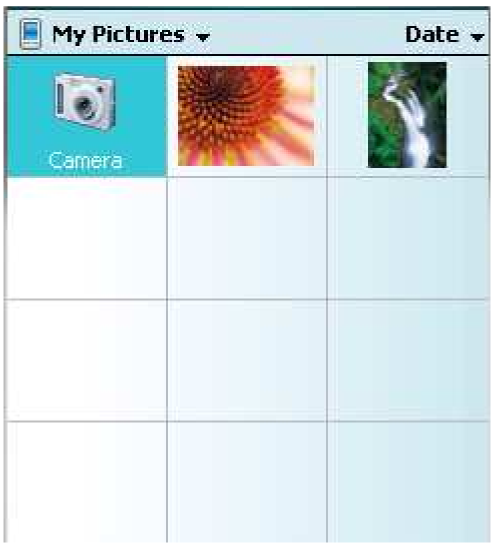
Semantics

In many cases the testability interface delivers information on the UI objects that are displayed by the software under test. The delivered information reveals the technical characteristics of the UI objects.

For example: for a list object, the testability interface will tell you how many items are listed inside the object, which item is selected, if the list is visualized by a list of icons or by a list of names, etc…

The testability interface will not deliver information on the semantics associated to the UI object.





As a human being, having experience with mobile phones, one can see immediately that the left figure is displaying information on a file system while the right figure is displaying a list of pictures.

One knows immediately also the actions associated to a selection of an object in these lists. Pictures can be opened and viewed in detail. Folders can be opened. Files can be opened by the associated program.

This information is not delivered by the testability interface.

In the QA process, the QA engineer needs to verify if the semantics of the software under test are matching the requirements. He is doing this by defining test cases that use the semantics of the application under test and that perform verification points against the requirements. In order to deal with the fact that one module of the software under test can match multiple requirements; the QA engineer creates a test framework so that he can re-use test scripts.

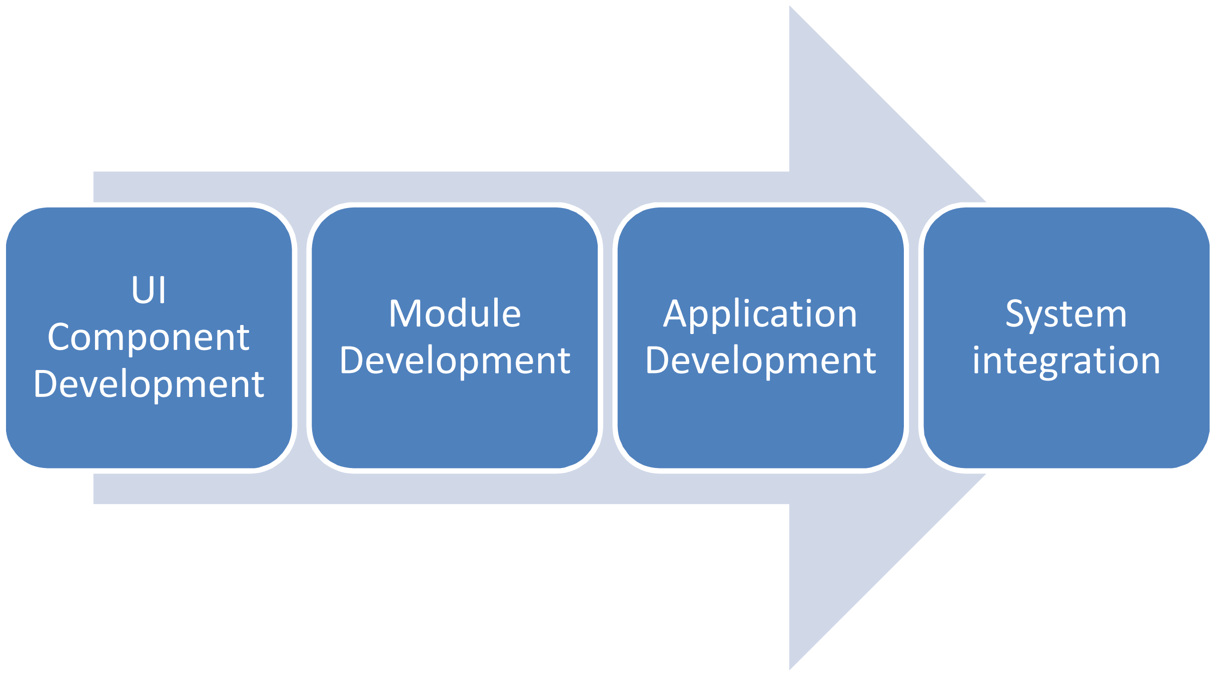
This framework that implements the automatic testing of the semantics of the software under test can only generate positive ROI if the framework is:

- structured
- easy to implement
- easy to maintain
- easy to expand
- complete
- reliable
- easy to reuse in multiple types of testing.

In order to meet the acceptable ROI of a test framework, this paper proposes following best practice: The re-use of the test effort following the applied implementation and design model.

Implementation processes

Following figure illustrates a common sequence of processes for building a software application with a graphical user interface:



The UI component is re-used in a module. Modules compose an application and this application is integrated into a system. Each phase results in building blocks that are re-used by the subsequent phases.

Each process has normally an associated QA phase. One will test the correct functioning of the UI component before it is released to the module builders. When the module is ready, one will test the correct functioning of this module. As soon as all modules are integrated in the application, the application testing phase will start followed by an integration test, system test and end-user acceptance testing phase. In contrast with the implementation process, in the testing world, there is little re-use of building blocks between the different phases.

Many times, each test phase is implemented independently from the previous test phase leading to inefficient use of resources and information. Best practice is to structure the test framework following the implementation and design processes so that testing effort is re-used throughout the development lifecycle and throughout the whole organization.

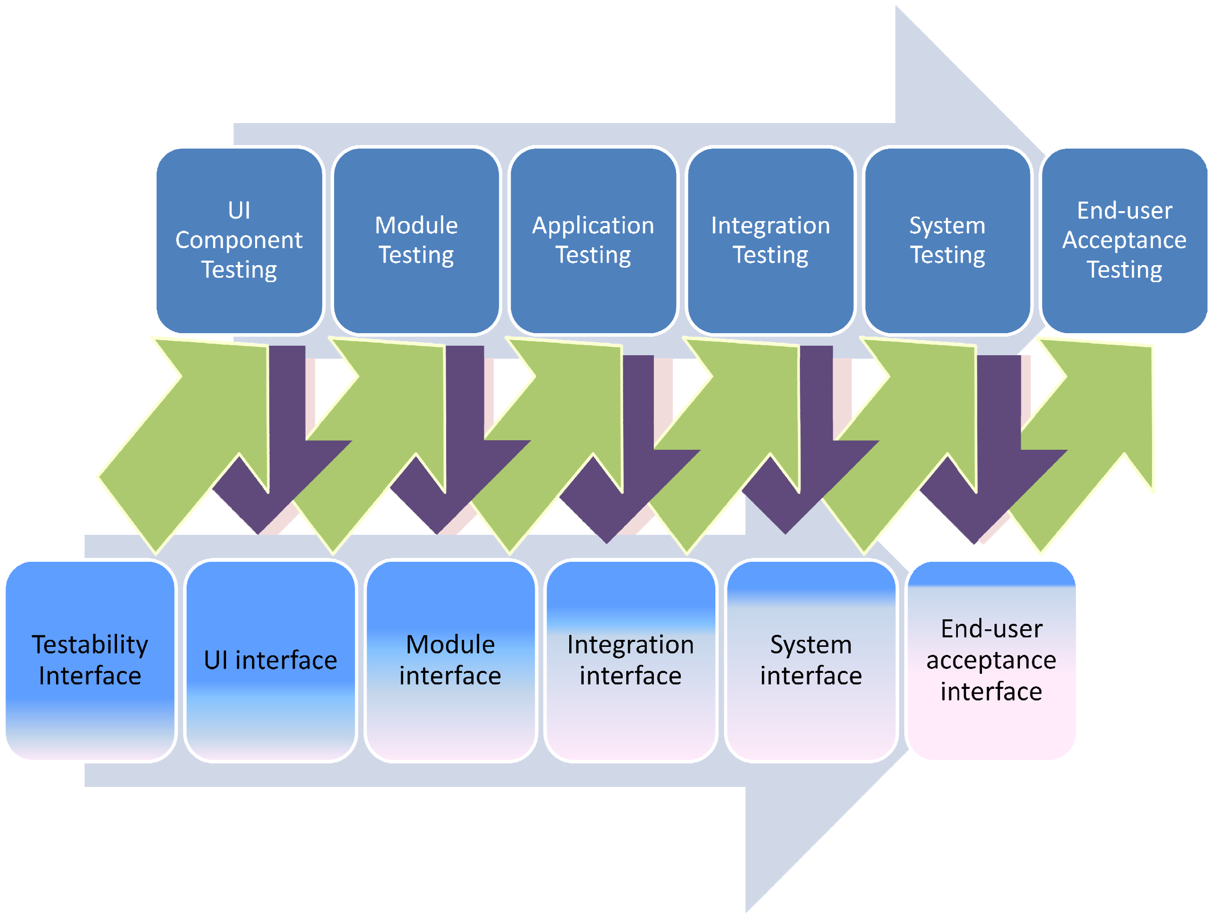
Ideal testing framework

The ideal testing framework is structured following the development lifecycle, re-uses information from the previous phase and exposes a new testing interface with a higher abstraction to the next phase.

The testing effort in phase XYZ is characterized by:

- A set of test cases that verify the requirements associated to the phase. The test cases are implemented by using the abstraction levels defined in the current and the previous phase.
- An execution test bed for executing the test cases. This execution test bed manages the test data, the hardware of the test lab, the dependencies and the pre-conditions and post-conditions of a test case execution.
- A diagnostic module that can be turned on to deliver additional information when a malfunctioning of the software under test is detected.
- An abstraction interface that will be used by the next testing phase and that exposes on a higher level the semantics covered in the current phase.

The testing effort for a new application, the different testing phases and the use and construction of the associated testing framework is illustrated by following figure:



The automated test cases in the UI component testing phase are scripted using the functionality exposed by the testability interface. At the end of the UI component testing phase, the testing framework needs to be extended by a higher level interface that exposes the semantics of the UI component and the associated test bed management without exposing the low level technical characteristics of the UI component. This higher level interface will be used to create the automated test scripts in the module testing phase. While scripting and executing the module test cases, an additional higher interface will be added in the testing framework to expose the semantics of the module and the associated test bed management. This interface is then used by the Application testing phase. The process is repeated until you reach the end-user acceptance testing phase. At this point, the testing framework will be complete.

By enforcing the definition of an abstraction interface for each testing phase, you enable the modularity inside the testing framework and by using the testing framework for creating the automated test cases you enable automatically the re-use of test assets.

By moving from one phase to another, the testing framework gets completed step by step. The functionality that is added to the testing framework in a specific phase is divided into two categories: functionality to check the requirements (blue color) and functionality to manage the test bed (pink color).

In the first phases, the QA engineer is completing the testing framework so that the semantics of the application under test are executed automatically, i.e. the framework is completed with action methods to drive the system under test, verification methods to verify correct functioning, synchronization and exception handling in order to deliver reliable unattended automatic execution, etc …

As soon as integration testing is passed, the testing framework has implemented a high level interface covering the semantics of the application under test. The people that are using in this stage the testing framework for implementing test cases do not need to have any more programmatic skills. The testing framework exposes a ‘natural to use’ environment to create tests by subject matter experts.

From integration testing on, the management of the data and the test bed for test execution get more and more complex. The testing framework needs to be extended now with functionality that manage this complexity and hides it to the end-users of the framework (pink color in the figure above).

To conclude, the test framework created in this way fits following requirements:

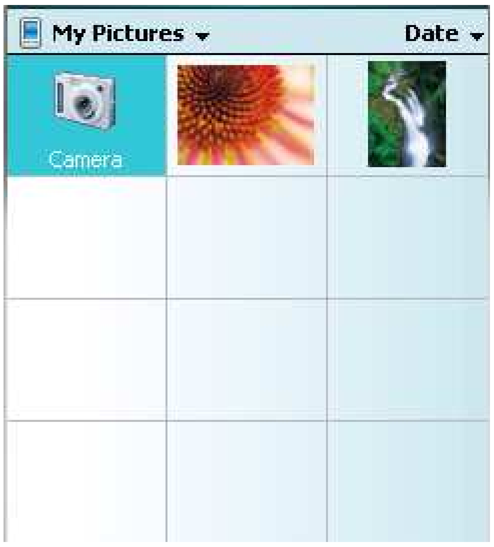
- The framework is *structured* so that changes in the software under test lead to identifiable and manageable changes in the test framework and created test cases. Minor changes in the UI interface of the application will affect only the UI interface and/or module interface implementation of the test framework and not the created test scripts!
  A major change in the UI interface of the application will affect the implementation and also the exposed functionality of the UI interface and the module interface of the test framework. Since the exposed functionality of the framework is changed, also the scripts created for the UI component and module testing phase will need to be modified. However script on application level, integration, system and end-user acceptance testing will not be affected and can be re-used as such for regression testing.
- The framework contains different abstraction levels for different kind of test cases so that the testing scripts are *easy to create and to maintain* in all phases of the software development lifecycle.
- By default, the objective of the framework is to *re-use easily* all testing assets, including the management of the test bed for test execution.

Design example of the testing framework

In this example, we will design and show a test framework for the contact application on a windows mobile enabled device.

UI Component testing

The automated test script for the UI component testing will use the commands from the testability interface. Following figure illustrate the SysListView32 UI object. Using this UI object you can visualize lists of items. An item can be visualized by text or by an icon. In the figure, icons are used.



The following automated test script checks if the horizontal spacing between the icons is respected:

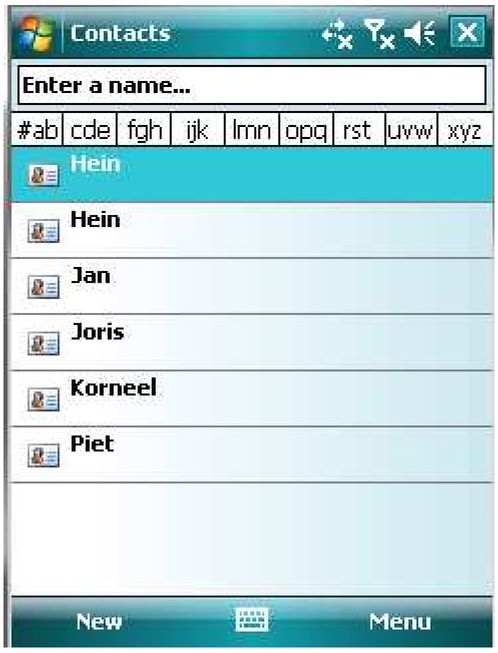
```
x = SysListView32.GetProperty( 2, “x”);
width = SysListView32.GetProperty( 2, “width”);
horizontalSpacing = SysListView32.GetProperty( 3, “x”) – x – width;
If ( horizontalSpacing <minimumSpacingMargin)
        Log( error, “Spacing is incorrect between the second and the third item”);
```

The SysListView32 object is an implementation of a general list object. An abstraction interface will be created with following content:

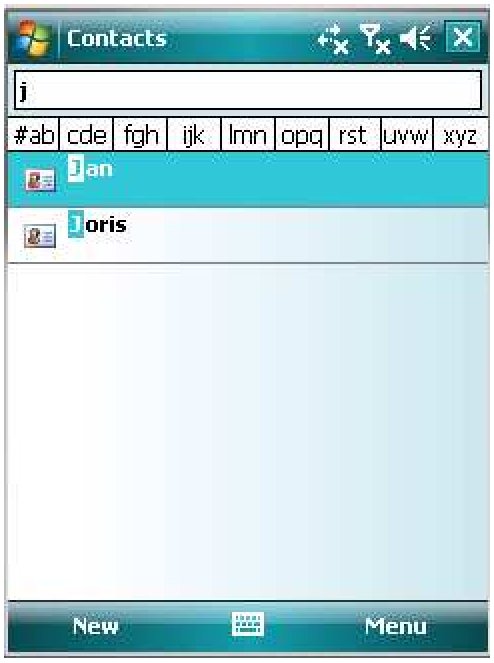
Objects		
	List	Represents a list UI object
	Item	Represents an item of the list
Methods		
	List.GetNumberofItems()	Return number of items in the list
	List.Select(item)	Selects the specified item so that it is highlighted
	List.Activate(item)	Select the item and activate it
Iterators		
	List.GetItems()	Iterates on the items inside a list

Module testing

The scripts for module testing will use only the commands from the UI-component abstraction interfaces. The following figure illustrates the window in which you can select a contact from the contact database. The contacts are showed inside a SySListView32 UI object.



One of the characteristics of this UI module is that you can narrow the list of displayed names by typing the first letters of the name of the contact you are looking for. After typing the letter ‘j’, the list shows only the names starting with a j.



This functionality is verified by following script:

```
List.Type( text)
For ( item in List.GetItems())
{
      If ( ! Item.GetText().StartWith( text))
            Log( error, “  item “ + item + “ does not start with “ + text )
}
```

If in a future release the list of contacts is not anymore a SysLIstView32 UI object, but another UI object visualizing a list, then there will be no need to adapt this script. Only the implementation of the list abstraction object will need to be changed. The modification in the UI layer of the system under test therefore leads into changes only in the UI-Component abstraction level of the testing framework.

Scripts created for this contact window can be encapsulated (re-used) for the next testing phase by creating the module abstraction interface:

Objects		
	ContactSheet	Represents the semantics and visualization of this contact window.
	ContactEditor	The contact detailed information view
Methods		
	ContactSheet.Hilite(contact)	Highlights the specified contact in the sheet
	ContactSheet.Select(contact)	Select the contact and opens the details
	ContactEditor.Set(field, text)	Sets the specified detail
	ContactEditor.Verify(contact)	Verifies the contact information fields for correctness
	ContactEditor.Close()	Closes the editor and returns into list view
Iterators		
	ContactSheet.GetContacs()	Iterates on the contacts inside the contact sheet

Application Testing

For writing the automated test scripts for the contact application, the tester will use only the immediate underlying abstraction levels and one of them is the ContactSheet abstraction.

Following test script checks if the data of a contact is correctly displayed:

```
ContactData contact = new ContactData (inputData);
ContactSheet.Select(contact.GetName());
ContactEditor.Verify(contact );
```

The script is very simple and implements following actions: a contact description is created from the specified input data. This contact description is used to select and open the contact in the contact sheet window. When the contact detail window is available a check is executed to verify if the displayed details correspond to the specified data.

Data used for input and verification will become more and more important in this phase and in the following testing phases. The testing framework will now also be extended with additional data management functionality.

The following abstraction interface will be made available:

Objects		
	ContactApplication	Represents the contact application.
Methods		
	ContactApplication.InitiateVoiceCall (text, field)	Make a phone call for the contact named by the text and using the number of the specified category (fixed, business, mobile1, etc ...)

	ContactApplication.ComposeSMS(text, field)	Start creating an SMS for the contact named by the text parameter and using the mobile number of the specified category.
Iterators		
	ContactApplication.GetContacts()	Iterates through all the contacts.
	ContactApplication.GetGroups()	Iterates through all the groups.

Integration Testing

The integration test cases are implemented by using the application abstraction layers. Following test scripts checks if a phone call can be initiated from the contact application:

```
ContactApplication.Run();
ContactApplication.InitiateVoiceCall ( text, “mobile1”);
if ( !PhoneApplication.isCalling())
        Log( error, “call is not initiated from the contact application for the mobile1 data field”);
```

Frequently used sequences of applications might be grouped into a new abstraction layer.

Objects		
	Call	Represents all actions to call a person
Methods		
	Call.CallUsingContactsApp (person)	Make a phone call by opening Contacts application and choosing “Voice call”
	Call.CallUsingNumber (person)	Make a phone call using Phone application and typing the number directly
	Call.CallUsingName (person)	Make a phone call using Phone application and typing the name directly

System Testing

The system test scripts use the application and integration abstraction layers. The script to test in depth the contact application with the SMS application will look like:

```
ContactApplication.Run();
ContactsInput contacts = New ContactsInput(inputData);
For (contact in contacts)
{
        ContactApplication.Select(contact);
        ContactApplication.ComposeSMS(mobile1);
        MessagingApplication.VerifyField( toField, contactName);
        MessagingApplication.SetField(“Body”, “This is a test”);
        MessagingApplication.Exit();
}
ContactsApplication.Exit();
```

The script implements following actions for all contacts given as a parameter. The SMS editor for a new SMS is opened. The body of the SMS will be specified and the SMS editor will be exited without sending or saving the information in order to return to the previous application.

Modeling and graphical notations for test scripts

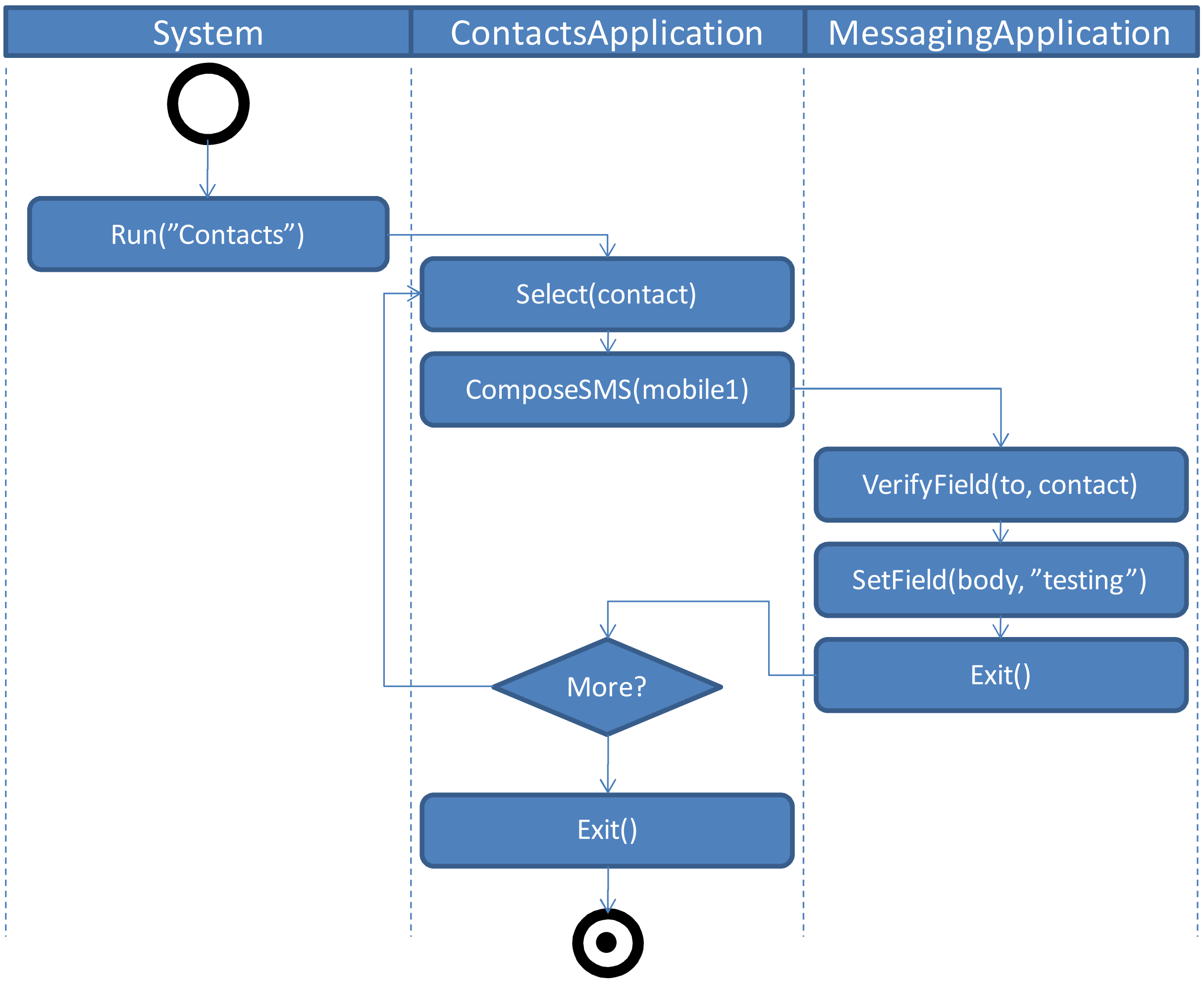
Since we are dealing with abstraction objects inside multiple abstraction levels, a graphical representation can be used to create the script(s). This will make the automated testing tool usable and accessible for the matter experts who are normally involved in the system and end-user acceptance testing phases. Expert users will most likely continue to use the "programmer view" even if the graphical representation is available.

Domain specific language for testing

We should not reinvent the wheel in graphical modeling of test scenario(s). We could use the Model Driven Architecture & Model Driven Development examples and notation for testing too. It should be quite straight forward to create a kind of Domain Specific Language for creating test scenarios. This way the scenario or model would be created graphically and the actual test script code would be generated from the model and executed independently.

Test scenario model

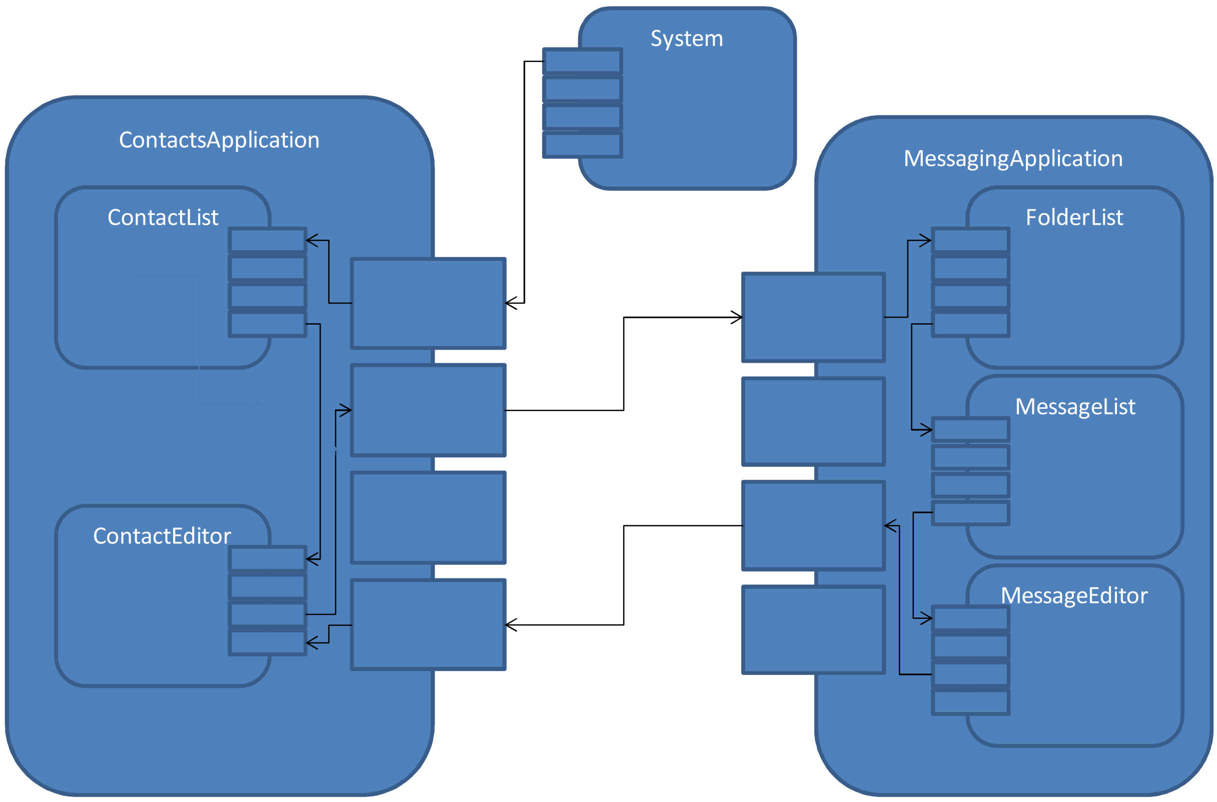
Modeling a scenario is typically done using flowcharts combined with actors that would be the abstraction objects in our case. This way we can model one test case or one test scenario defining the sequence of actions, the objects involved in the actions and the branching of the scenario. Below there is a simple example of a flowchart that describes the test scenario we used in one of the examples above.



In the flowchart the sequence typically goes from up to down. The swimlanes represent the actors which are test abstraction objects in our case. Actions/steps inside the flowchart map to methods of the abstraction objects.

Software under test model

Instead of writing out each test scenario explicitly, one should also be able to model the full system and then create out of this model automatically test scenarios. A known technique for doing so is the Model Based Testing technique.. The Model Based Testing relies on a graphical representation of the states and possible transitions between these states. Transaction can be characterized by probabilities and conditions. Since the model covers all states, it can get quite big. Therefore it would be useful to re-use the abstraction objects so that one can compose the model on a higher level re-using lower-level models using the underlying abstraction objects. Following diagram illustrates who re-using the abstraction layers of the testing framework allows you to create easy to use diagrams for model based testing techniques.



In the above model following abstraction levels multiple abstraction levels are shown: The ContactsApplication represents the application level and ContactList represents the module level. If the designer tool of the testing framework supports nesting it would be possible also to create new higher level components visually by using lower level components as building blocks.

Conclusion

Implementing a test framework with abstraction layers for each building block of the application under test results in a framework can deal with changes for regression testing, that is fit for cross-platform testing and that generates an acceptable ROI.

The described testing framework optimizes also as much as possible the re-use of test assets and test automation scripts inside the testing cycles so that ROI is also generated for testing just one release of the application under test (non-regression testing).

If followed thoughtfully, the abstraction is coming to a level so that instead of scripting the automated test cases, one can use a graphical flow diagram to create the test cases for the system and end-user acceptance testing phases. Using this graphical representation, you will be able to involve directly the subject matter experts in the creation of the test cases which will lead to higher quality test cases at a lower cost.

The testability interface is the basis for the whole testing framework. It is important that this testability interface is complete so that framework is built on strong foundations.

Jamo Solutions in brief

Jamo Solutions aims to promote testability in the area of test automation and application monitoring. The company mission is to work together with the market players to increase the completeness of the management of the end-user experience. Jamo Solutions develops, markets and sells tools and consultancy that extend the capabilities of best-of-breed testing and monitoring solutions for end-user experience measurement and management.