TRICENTIS

Model-Based Test-Automation

The use of a non-graphical model as an enabler for effective test automation

Dominik Weissboeck, John Smith

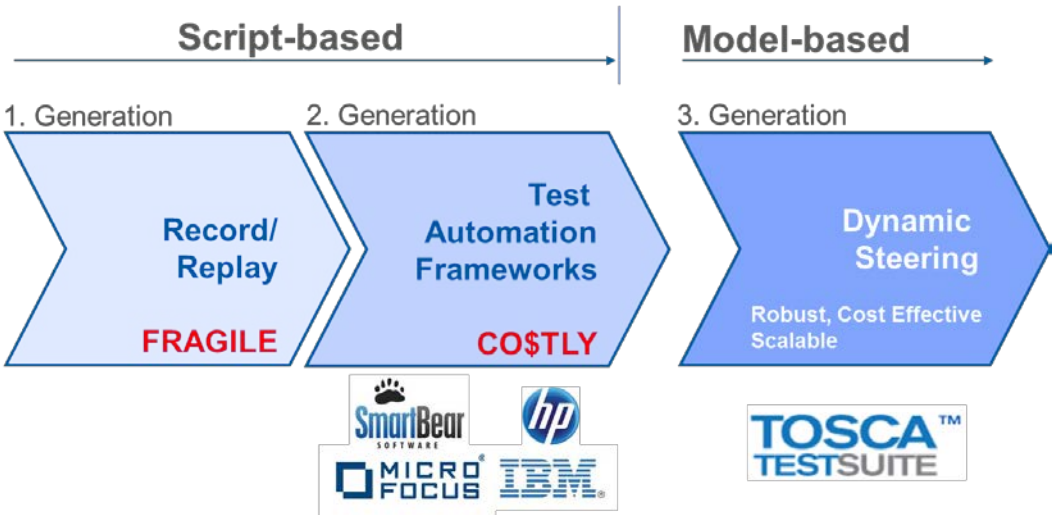
4/3/2012

This whitepaper describes how TOSCA uses a non-graphical model as an enabler for efficient test automation. The TOSCA model is derived from the software build itself (in particular its interfaces) and presents an abstract interface catalogue (system model), making it possible to write test cases that are independent of the choice of underlying software technology and that are maintained in a human-readable “business” language.

Contents

Executive Summary

TOSCA™ is a new, third-generation technology using a **model-based approach**, unlike first and second generation technologies such as record/replay tools, or tools that require test automation frameworks (script-based technologies).



Some of the **core business benefits of model-based testing** over script-based testing are:

- ✓ **Lower Total Cost of Ownership**
- ✓ **Speed to Market**
- ✓ **Improved Quality**
- ✓ **On Demand Staffing**

This whitepaper describes how TOSCA uses a **non-graphical model** as an enabler for efficient test automation. The TOSCA model is derived from the software build itself (in particular its interfaces) and presents an abstract interface catalogue (system model), making it possible to write test cases that are independent of the choice of underlying software technology and that are maintained in a human-readable "business" language.

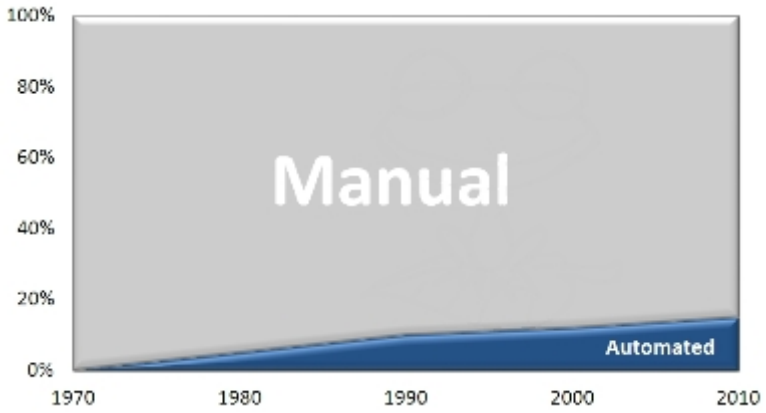
As such **TOSCA eliminates the maintenance issues inherent in first and second generation tools** and elevates test automation to a business discipline (for an impartial view see Appendix B: What GARTNER™ says about TOSCA).

Why Model-Based Testing?

In the year 2002 TRICENTIS developed the hypothesis that if we are ever to achieve robust, low maintenance test automation, it will require a dynamic approach that entirely separates the test logic from the automation logic. In other words, only if we remove the tight coupling between the test logic and the automation logic and only if we allow for non-technical experts to create and, ***more importantly****, maintain both the test logic and the automation logic, will we achieve higher test automation coverage.*

Why did TRICENTIS take a new approach to test automation? In particular, why did we think we needed to add this new concept to test automation - different from Record/Replay and Test Automation Frameworks?

The answer is that even after 40 years of test automation technology presence in the market, on average only 15 out of 100 regression tests are automated in Western Europe…



Source TRICENTIS Customer Survey 2008

…and only 13 out of 100 in Australia.

Source K.J.Ross & Associates – Industry Benchmark 2009/10

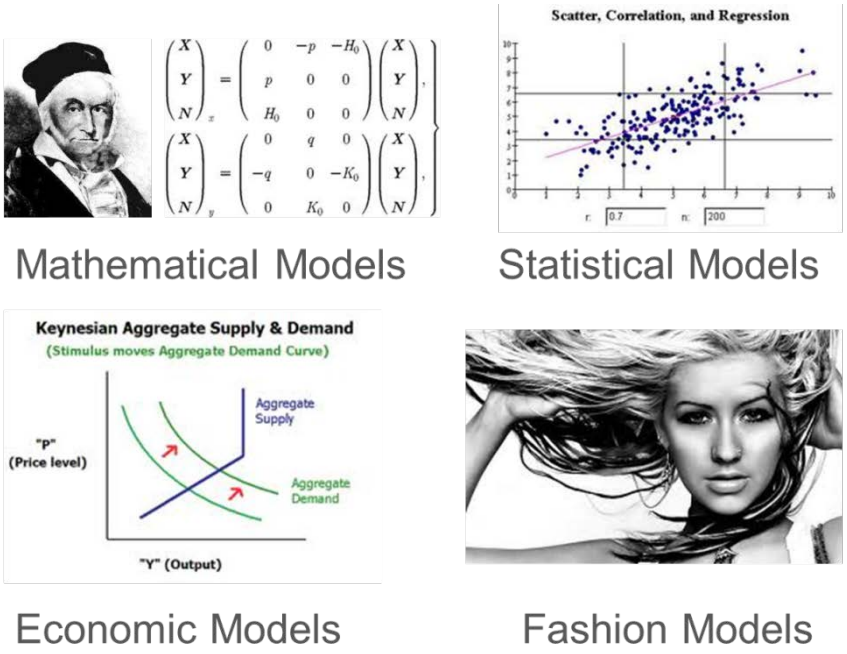
Why has test automation take-up been so slow?

Previously, with script-based technologies, your test cases were either rigidly integrated with the automation code (as in **Record/Replay)**, which made test cases very **fragile** under change, or you had to develop a **Test Automation Framework**, which was **time-consuming**, **costly** and **not very scalable (**particularly across heterogeneous technology environments). Consequently test automation coverage is poor across organisations (for details see chapter: “Three generations of test automation”).

Models in Context - what is a model?

Before we start explaining the differences between script-based and model-based approaches to test automation, we would like to address an issue about models in software engineering, particularly among interest groups in model-based testing, where a misconception that software models have to be graphical has arisen.

Models are ubiquitous in science, mathematics, engineering, business, economics and even fashion (see illustration below), and the use of the word ‘model’ so commonplace that readers are usually expected to understand from context what is meant.



Models

In software engineering in particular, models have proliferated to the point where ‘model’ is so overloaded a term that it needs qualification to have meaning. Indeed, as software has increased its influence on every aspect of life and business, systems thinking and models have come with it: we

have business models and enterprise models to add to requirements models, design models, process models, and more.

When we set out to create software to perform complex tasks, we do not simply sit with a blank sheet of paper and write line-by-line the instructions to a computer to perform the desired task. Instead we begin by describing what we want the software to do, using language appropriate to the domain – for business, we will use language that the business owners and users will understand – and then take that description through multiple levels of abstraction, elaborating and transforming along the way, until we have expressed it in terms a computer can execute.

To assist us in writing these descriptions, which, although abstract, can themselves be complex, we may use *diagrams* to augment text, and, if the notation is formalized, we can say we have created a *graphical* (or *visual*) *model* of some aspect of the software. Such diagrams help in creating and *communicating* a shared understanding of desired software behaviour or structure. UML®, from the Object Management Group (OMG®), is an example of a formal notation, combining text and diagrams, which is in widespread use.

Such models are aimed primarily at software development and have most utility during the requirements elicitation and design activities of the software development process; once development is complete, there may be reason to construct other types of models, such as execution models to examine the behaviour of software at execution time at a low level. These models may or may not have a graphical component.

Models in general will use whatever representation is appropriate to their intended use: they are, after all, descriptions of whatever is being modelled: architects will often build scale three-dimensional models of their designs, mathematicians will (not surprisingly) use mathematical notation to present their models, and in software development, a variety of model types will be found, each suited to its purpose. For programming details, a text-based approach is usually adopted because graphical notations are not as good; for broad-brush software structure description, a graphical approach is better – in short, models do not have to be graphical.

The development team needs to take a hard look at how models will be used in their development process, how elaborate the models will become and the purpose of such elaboration. The maintenance of detailed models alongside an evolving system is effort-intensive, and the team needs to have clear reasons for doing it, and understand the cost/benefit trade-off. **See the sidebar under "Appendix A: The Evolution of Modelling" below for a deeper discussion of these topics.**

Graphical models are very appealing and effective for use by humans in the creation and communication of specifications and designs of all kinds – including software. For machine to machine interaction (of which automated testing is an example) they are simply unnecessary – although a machine can of course be programmed to parse a diagram.

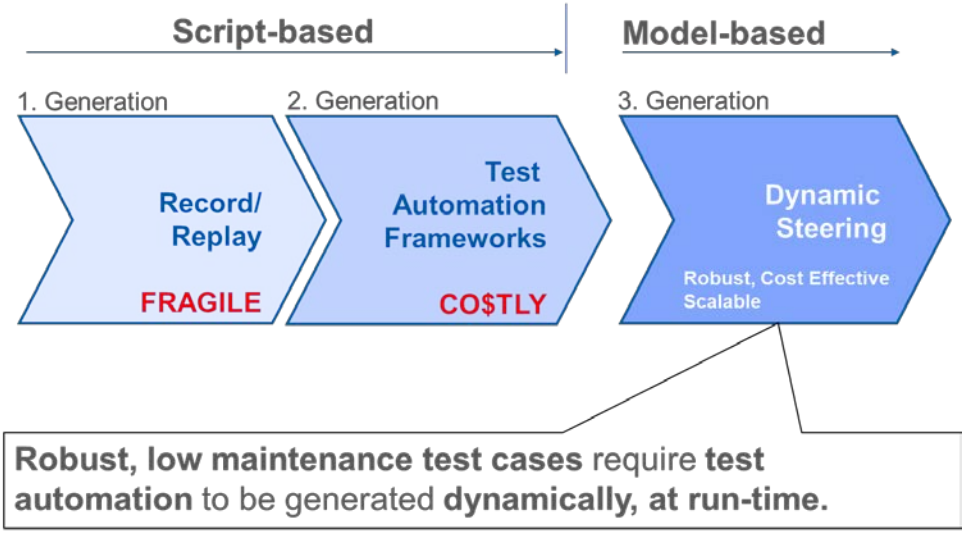
To have derived any sort of graphical model for this purpose would have added no value, because the model is *not* intended to automate the *production* of test cases, but their execution.

TRICENTIS does recommend the use of the requirements specification (and any associated models) as a major driver for test case *definition* (but not automation). These models may include UML Use

Cases, with the flow of events defined using more formal techniques such as Activity Diagrams and State Transition Diagrams, as well as text.

The TOSCA automation model certainly qualifies for the description 'model'– albeit a high-fidelity model – because it is an abstraction of the actual system and not part of its end use. However, its usage is highly efficient: it can be re-derived quickly and with low effort whenever necessary.

Three generations of test automation

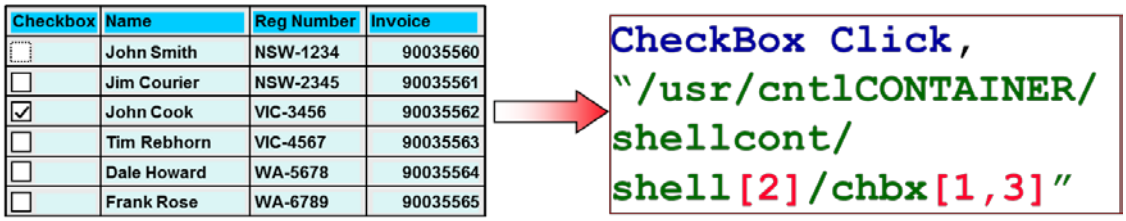


Three generations of test automation

Why is Record/Replay (Capture/Replay) FRAGILE?

*The problem is that in script-based technologies using a record/replay approach, your test cases are rigidly integrated with the automation code, and this makes test cases very **fragile** under change.*

For example, the task is to "Select the invoice for John Cook" in the table below.



Checkbox	Name	Reg Number	Invoice
☐	John Smith	NSW-1234	90035560
☐	Jim Courier	NSW-2345	90035561
☑	John Cook	VIC-3456	90035562
☐	Tim Rebhorn	VIC-4567	90035563
☐	Dale Howard	WA-5678	90035564
☐	Frank Rose	WA-6789	90035565

```
CheckBox Click,
"/usr/cntlCONTAINER/
shellcont/
shell[2]/chbx[1,3]"
```

In a Record/Replay approach, a task such as “Select the invoice for John Cook” in the table in the illustration above, is translated to code, which simply states “click a checkbox in column #1 and row #3” in the table.

Situation 1: What happens if the layout of the table changes?
What if the checkboxes are moved to the other side of the table?

Name	Reg Number	Invoice	Checkbox
John Smith	NSW-1234	90035560	☐
Jim Courier	NSW-2345	90035561	☐
John Cook	VIC-3456	90035562	☑
Tim Rebhorn	VIC-4567	90035563	☐
Dale Howard	WA-5678	90035564	☐
Frank Rose	WA-6789	90035565	☐

The automation code breaks “technically”, i.e., it will not be able to locate any checkboxes in column #1 that can be “clicked” and will therefore fail to execute.

Situation 2: What if the test data order is changed?
What if John Cook now appears in row #4 instead of row #3?

Checkbox	Name	Reg Number	Invoice
☐	John Smith	NSW-1234	90035560
☐	Jim Courier	NSW-2345	90035561
☑	Tim Rebhorn	VIC-3456	90035562
☐	John Cook	VIC-4567	90035563
☐	Dale Howard	WA-5678	90035564
☐	Frank Rose	WA-6789	90035565

The automation code fails semantically, i.e., the code will execute without technical failure but it will not select the invoice for John Cook as originally intended. The objective of the test won’t be met, and thus result in a semantic failure of the automation code.

Conclusion: Whenever the layout or the test data change, the automation code will break - either technically or semantically. Consequently, even with these simple examples, we can see that Record/Replay adds little value when automating test in software development, where change is almost certain. And we have not even started to touch on some more serious and interesting challenges in test automation, for example, how do we address the issue of a change in the table identity?

In summary, for Record/Replay, we have:

- Advantage: quick to develop
- Disadvantage: not stable enough to add value during software development

Why are Test Automation Frameworks CO$TLY and TIME CONSUMING?

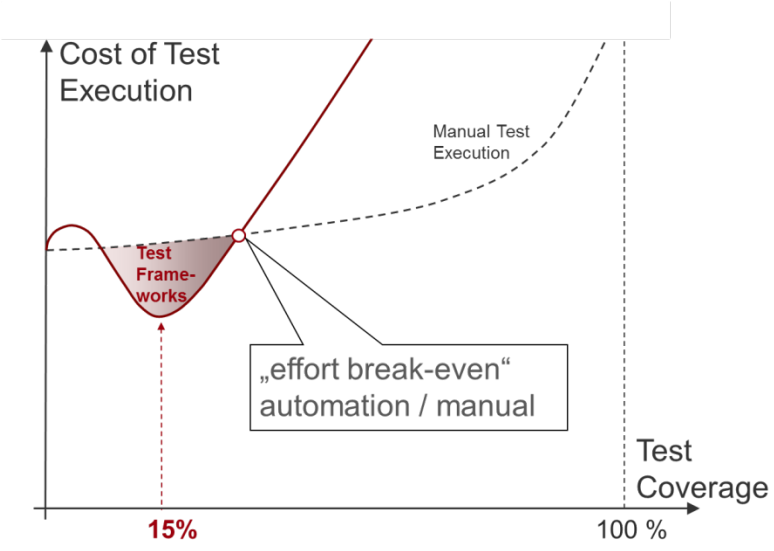
*Although Test Frameworks address the issues of Record/Replay to some extent, they are a very **costly** alternative, because testing then requires its own parallel software development project.*

In order to address the shortcomings of Record/Replay – the fragility of test automation scripts and consequent maintenance burden – organisations have started to develop test automation frameworks, in which the scripts are enhanced through **programming.** That is, automation scripts are **modularized** and **generalized**.

Modularised means that the automation script is broken down into function-calls (snippets of code) that can be reused, such as a "Login" function or "Enter Person Details" function. These function calls are then labelled with keywords, and these are commonly referred to as keyword-driven frameworks.

In addition, test data is extracted from the function calls and put into a data source, thus allowing for manipulation of data outside the function call leading to a more **generalised** use of the function calls through use of different test data.

Although Test Frameworks address the issues of Record/Replay to some extent, they are a very costly alternative, because testing then requires its own parallel software development project.



Effort break-even using test automation frameworks

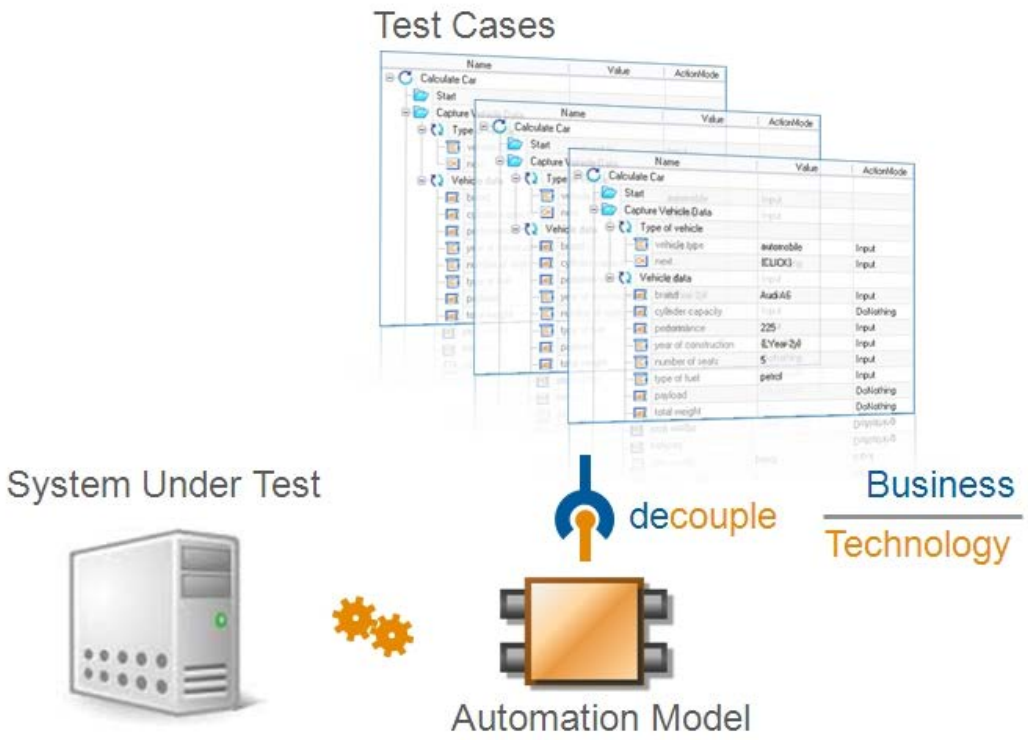
Consequently the incremental cost of increasing your test automation coverage quickly exceeds that of manual execution. The break-even effort point between manual testing and automation is reached in a test automation level range from 15% to 25%, across heterogeneous IT environments – typical for most organizations!

The **root problem** with both the Record/Replay and the Test Automation Framework approach is the automation code itself – it is either fragile or requires a high level of maintenance!

What is different about model-based test-automation?

Model-based test automation **decouples** the test cases from the system's underlying technology. A test case in model-based testing is created and – more importantly – maintained in a human readable language (e.g. plain English) at all times, before and after test automation. In this environment, test cases will never be converted into automation code.

As a matter of fact, in model-based testing there is only **one source of truth for test cases**, regardless of whether they are manual or automated.



Separation of concerns and encapsulation

In software development, **decoupling refers to the careful separation of concerns**, in our case the separation of the test logic (the test cases) from the automation model. The test cases should not

depend on the automation model, nor should the automation model be tied to the test cases, but should be an asset that provides value for automation beyond testing, for example, in automated data handling or system navigation.

In model-based testing the automation model – and recursively its components – do not have any implied test logic (no sequence, no data), as opposed to a **Record/Replay** approach where the test logic is rigidly integrated with the automation code, or even **Test Automation Frameworks** where keywords, which are essentially labels for function calls (snippets of code), often encapsulate some test sequence – unless a very mature and highly disciplined development process is in place and keywords are modularised to the lowest level (individual controls on a GUI or individual methods on a non-GUI).

In model-based testing it is only at runtime, that is, at the time of test execution, where test logic is injected into the automation model.

This rigorous decoupling of the automation model from the test logic in model-based automation, through separation of concerns and encapsulation of these concerns (the same principles used in object-oriented programming), has several significant benefits:

- **Reuse**: The components in model-based automation offer a high potential for reuse, for any automation purpose, testing or otherwise. Rigorous decoupling increases:
    i. the reuse of the automation model and its components in different test scenarios, or for other automation purposes, such as data creation or system navigation
    ii. the reuse of test cases against different system technologies, for example running the same test case against a thin client (HTML client in a browser) and a thick client (for example, one developed in Java or .NET)

- **Robustness, Flexibility and Low Cost Maintenance**: In model-based testing a change in the test logic (sequence or data) does not require maintenance of the automation model. At any point in time you are free to change the flow, the conditions, the sequence, the test data, etc. in a test scenario without requiring maintenance effort on the automation model. On the other hand, maintenance effort due to changes in the system's technology is minimized through encapsulation – you update the corresponding component in the automation model, and all test cases are up to date. Thus, model-based testing eliminates maintenance effort that is inherent in other test automation approaches.

- **One-Universe / One- View**: model-based testing provides you with **one universe** for:
    i. **Manual and automated testing** – there is only one source of truth for test cases regardless of whether they are executed manually, semi-automatically (in any order of manual and automated test steps within a single test case) or fully automatically.

    ii. **GUI and non-GUI testing** – a component of the automation model can be a proxy of a graphical user interface as well as a proxy for a non-graphical user interface, such as a web-service, a database table or messages on a message broker.

iii. **Different Technologies** – components of the automation model can interact with different technologies, such as HTML/AJAX, FLASH, Web-Services, JAVA, .NET, C++, VB, MAINFRAME, DELPHI, POWER BUILDER, DATABASE (SQL), to name a few

iv. **Bespoke systems and packaged applications:** modules can be derived whether the system has been developed in-house (bespoke systems) or provided by a vendor (packaged applications such as SAP or SIEBEL)

v. **Single System testing and end-to-end business process testing**: the scope of a test case can range from a simple verification on a single system to complex end-to-end business process test scenarios across multiple systems, because automation components for GUI and non-GUI, as well as any technology mix, can be used within the same test case.

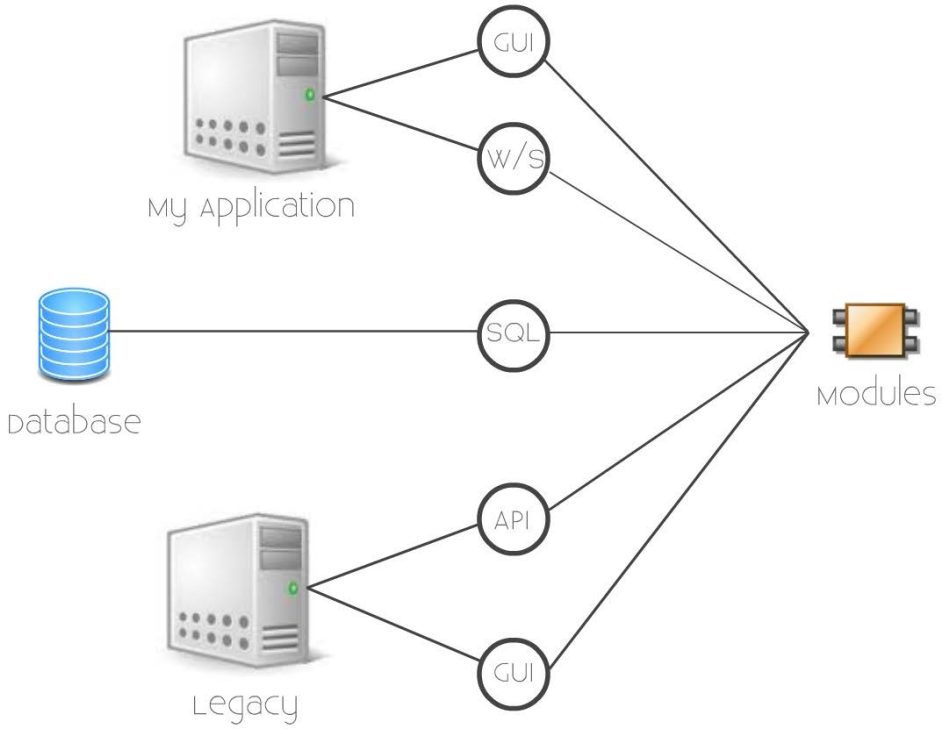
.

- **On demand staffing**: Automated test cases can be created by dragging and dropping components of the automation model onto test cases and then defining the data and action against it. As a result, automated test cases can be created and maintained by business analysts or test analysts without programming skill – not just by test automation specialists.

- **Speed to market**: The above-mentioned *flexibility*, *low maintenance* and *on demand staffing* benefits, together with a fast learning curve for model-based testing (there is little difference in the application of model-based testing on GUI or non-GUI or differing technologies), facilitate on-demand and fast-paced change.

How do you create the automation model in TOSCA?

You can think of TOSCA's automation model as a reverse-engineered interface catalogue of the system-under-test, in which modules (components of the automation model) are proxies of the interfaces (GUI or non-GUI, technology A or technology B).



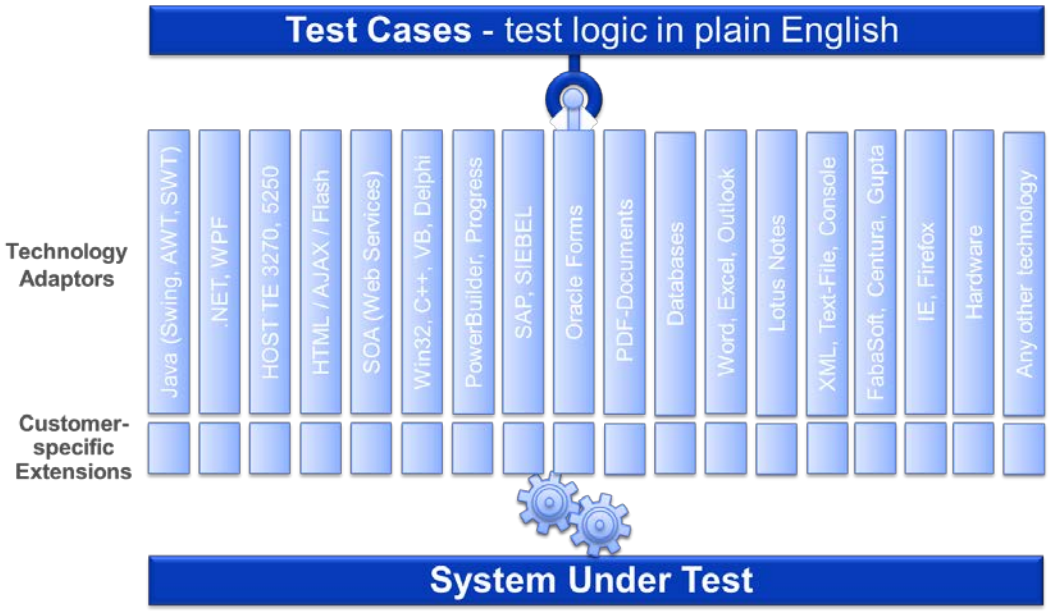
Modules are proxies of interfaces

To reverse-engineer the interfaces into modules, TOSCA provides a set of technology adaptors/engines (see illustration below). These technology adaptors are transparent (invisible) to the user but allow the user to point TOSCA's wizard at an interface, scan its content and create the module.

The modules can then be used via drag and drop either to assemble a test case from scratch or to link modules to an existing manual test case so it can be executed automatically.

Furthermore, with the wizard, not only can a TOSCA user reverse-engineer an automation model from a system's interfaces (GUI and non-GUI), they can also synchronize an existing automation model with a system's interfaces in case of technology changes, for example, when a new version of a system has been developed and needs to be tested.

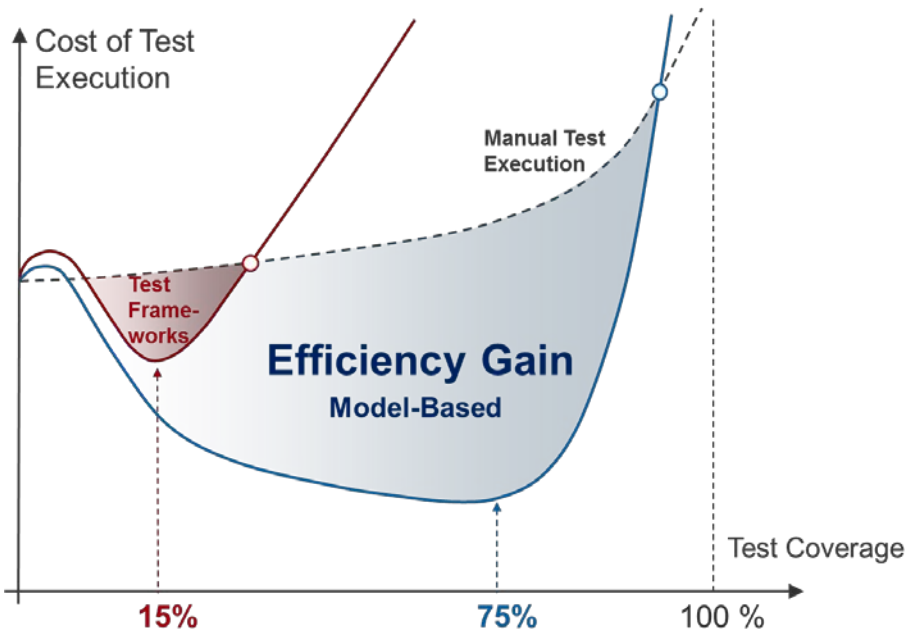
TOSCA's scanning and synchronising capabilities automate the creation and maintenance of an automation model with significantly less time and cost than programming a test automation framework.



Technology Adaptors

The Bottom Line

The bottom line advantage for model-based test automation lies in its elimination of maintenance issues inherent in Record/Replay and Test Automation Framework approaches, and consequently its ability to take test automation coverage to much higher levels, significantly lowering the cost and elapsed time for testing.



Efficiency gain of model-based testing over script-based testing

Appendix A: The Evolution of Modelling

There is an ongoing effort within the OMG to establish Model-Driven Architecture® (MDA®) as its flagship specification: MDA would establish standard guidelines and specifications to provide a common vendor-neutral framework for application development and encourage interoperability across major hardware and software infrastructure platforms; models are foundational for MDA, not merely documentation or blueprints for humans to write code, but are themselves enactable or executable – or automatically transformable into representations that are. MDA relies on the availability and acceptance of rigorous model specification capabilities, for example the MetaObject Facility (MOF) and UML. So because UML has such an extensive graphical (visualization) repertoire, it is not surprising that any mention of model in a software engineering context elicits an assumption of a visual model. That assumption is unwarranted: models use whatever representation is appropriate for their purpose and the aspect of the system they represent. Even within UML, text has always had an important role (in Object Constraint Language, OCL, for example), and, as we shall see, an even more important role in the evolving UML.

For any model to be useful in an MDA environment, it must therefore have specifications precise enough for execution – which is not possible with standard UML. This is also true for any attempt to use models to drive testing – either as direct targets-of-test (in which case the model has to execute either in the real target environment or with some kind of virtual machine), or as a means of deriving test cases for later use with the real software (itself derived from the model).

The motivation here is an assertion that test cases derived from the model are:

- not arbitrary nor at the whim of the tester, but a reflection of the desired/actual (in the model) behavioural semantics
- stable (because the model changes infrequently once development is underway) and at a higher level of abstraction

Is this possible? The answer is yes, it has already been done – but is it cost-effective? The models have to be built with high fidelity with respect to the final system, and with enough precision to be able to derive tests, and this is a lot of work. As things stand, we still have to produce code after this modelling effort: MDA has been dormant for the last few years, the vision remains unfulfilled. The OMG has recently breathed life into the idea of executable UML with the release of Foundational UML (fUML) and the Action Language for fUML (Alf). The fUML standard specifies precise semantics for an executable subset of UML, and the (Alf) standard specifies a textual action language with fUML semantics.

It is interesting, but not surprising, that the technology (Alf) that enables precise specification for execution is text based not graphical, as Ed Seidewitz said in his presentation to the OMG [ES220311] “graphical modeling notations are not good for detailed programming”. Graphics are good at a high level of abstraction and assist human understanding; for detail and machine understanding, text is better.

If this OMG effort were to be successful and found acceptance in the software community at large, then have we not simply moved the programming language up one level of abstraction? Now we program in fUML and Alf, and tools then transform our programs into machine executable representations. The 'model' has become the system – at least in the same sense that a body of Java or C++ source is the system! Model-driven testing is then simply… testing. Any test cases derivable from the 'model' are derivable from the generated system – because this generation process is mechanical, the generated code is faithful to the 'model'. TOSCA can help automate this derivation, with the added bonus that the test cases, themselves assembled from a model of the system's interfaces , are expressed in business-facing language.

If you share the OMG's vision that the success of modelling notations and technology (UML in their case) as practical system *generators,* as envisaged with MDA, depends on the adoption of technologies like fUML and Alf and the creation of tools that support them, then you will agree that model-driven testing (in the sense of producing executable tests derived from visual models) becomes redundant. There is still a place for test case generation from visual models while they are still high-level (in the case of fUML and Alf, before a wealth of Alf detail is added), as a bridge from the requirements for the software to outlines of the test cases that will be further elaborated once the system exists.

It is difficult to see the sense in producing a *non-generative* visual model of sufficient detail that it would be useful for detailed test case generation for test automation – and then continuing coding the system, most likely in a different language – unless you enjoy doing things twice.

[ES220311] Programming in UML: Why and How *Ed Seidewitz, Model Driven Solutions, A division of Data Access Technologies, Inc.* June 2010

Appendix B: What GARTNER™ says about TOSCA

Gartner™ has examined TOSCA™ and says: *"We believe that model-driven testing approaches will become the dominant format in five years, and that TRICENTIS is playing a leading role in this trend. Getting testing automation to work has been a great challenge. Often, tools will demo well, but in daily use they fall flat in the ability to create test automation that keeps up with software changes. This leads to very low test automation rates and high costs to create and maintain tests.* ***TRICENTIS, with its TOSCA Testsuite product, has broken the mold to enable improved productivity*** *in the creation of tests and in the ability to maintain these tests."*

Gartner™ continues: ***"Who Should Care****: Testing organizations that have been struggling to get the desired return on investment (ROI) from existing tools should investigate emerging tools that take model-driven approaches."* Source: Gartner™ Cool Vendor Report G00175440 (c) 2010.