Faculty of Engineering of the University of Porto

An overview of testing automation techniques

Pedro Tavares

November 3, 2017

Contents

1 Introduction

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing can also provide an objective, independent view of the software quality and performance that allows the business stakeholders to assess and understand the quality of software implementation. Software Testing has also been considered as a time consuming activity that calls for enhanced and powerful test techniques with the intent of finding software bugs. In this paper, we compare some of the techniques and strategies used to take full advantage of testing automation.

1.1 Classic Testing Processes

1.2 Manual Testing Process

1.3 Capture/Replay Testing Process

1.4 Script-Based Testing Process

1.5 Model-Based Testing Process

2 Data-Driven Testing

Data-driven testing is a scripting technique that stores test inputs and expected outcomes as data, normally in a tabular format, so that a single driver script can execute all of the designed test cases [1]. Usually, testing scripts that don't follow a data-driven approach, have data attached into them. When test data needs to be updated the actual test script must be changed as well. This might not be a big deal for the person who originally implemented the script but it may become a problem to a test engineer that has less programming experience.

Data represents a large part of system and test specification. By considering how data is defined and used during testing we can increase the opportunity of reuse as well as reduce future maintenance costs for subsequent releases of the system being developed. That's what make data-driven testing extremely useful since one single test script can be used to run different tests, reducing the number of the overall test scripts needed to implement all the test cases.

A suitable scenario where data-driven testing can be applied is when two or more test cases requires the same instructions but different inputs and different expected outcomes. The refactoring would result in one single test script and a shared input data file.

2.1 Test Data

Test Case	Input 1	Operation	Input 2	Result
Addition 01	5	+	5	10
Addition 02	5	+	10	15
Subtraction 01	10	-	5	5
Subtraction 02	10	-	0	10
Multiplication 01	5	*	5	25
Multiplication 02	5	*	0	0
Division 01	10	/	2	5
Division 02	10	/	10	0

Table 1: Data-driven test data with mathematical operations.

2.2 How to store test data?

Data-driven testing calls for tabular data and the first instinct is to use spreadsheet programs to edit it [2]. Spreadsheet files like comma-separated-values (CSV) are handy because they are very easy to parse but unfortunately the data becomes inconsistent when edited by two or more different spreadsheet programs. But the biggest problem with storing test data into any kind of flat file is scalability [2]. If for example test data is created and edited in several workstations and used in multiple test environments it gets hard to have the same version of the data everywhere. Moving the data files to a central database, backed up by a web-based interface in order to manage the test data, is a suitable solution when scalability is a requirement. This way, the driver scripts can read test data directly from the database instead of parsing CSV files before running the test suite.

2.3 Processing Test Data

Implementing a script for parsing data-driven test data can be surprisingly easy either using data files or databases [2]. Data is processed line by line and split into cells that represent the test inputs making the data available to be used in the test scripts. The main drawback is that the parser will always be limited in extensibility and it will crash if the test data format changes. Whatever design decisions are made about the driver script, the test data must reflect to it [1]. This means that if we change either the driver or the format of the data, we need to make sure they are always in sync.

Listing 1: Example of data-driven driver script.

```
const TestData = require('./testData');
const Calculator = require('./calculator');

TestData.testCases().forEach(function(testCase) {
  const input1 = testCase.input1();
  const operation = testCase.operation();
  const input2 = testCase.input2();
  const expected = testCase.result();

  execute(input1, operation, input2, expected);
});
```

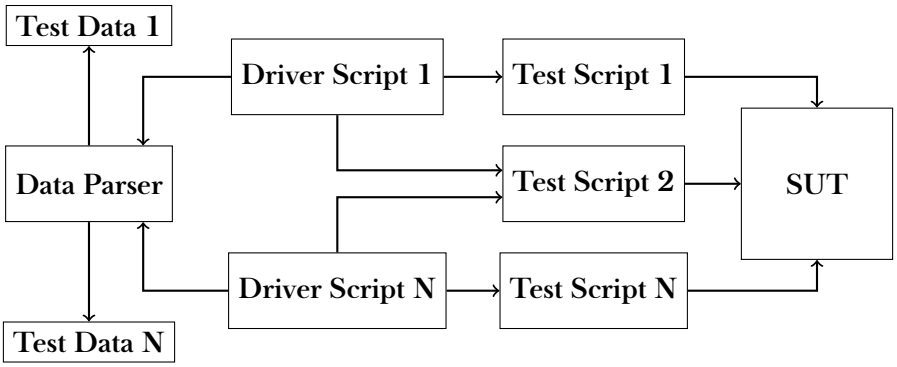

Figure 1: Data-driven system design approach.

In this data-driven approach there are a few elements that are crucial and must be implemented:

- **Data Parser**: It is responsible for parsing the test data, either from a CSV file or a database table. Its main task is to process test data and return it to the driver script providing an easy and neutral access to the test data.
- **Driver Script**: It is responsible for driving the test execution process using the testing functionality provided by the test libraries, that interact with the system under test, and the test data, that is read by the data parser.

2.4 Advantages of data-driven testing

A major advantage of the data-driven testing is that the format of the data file can be tailored to suit the testers needs [1]. The data file can contain comments that the script will ignore but that will make the data file much more understandable and, therefore, maintainable even by non-technical people. The easier it is, the quicker and less error prone it will be.

With data-driven testing we can really start to benefit from test automation [1]. It is possible to implement many more test cases with very little extra effort since all we have to do is specify a new set of input data and expected results for each additional test case.

2.5 Disadvantages of data-driven testing

The biggest limitation of the data-driven approach is that all test cases are similar and creating new kinds of tests requires implementing new driver scripts that understand different test

data. Thus the test data and driver scripts are so strongly related that changing either requires changing the other. In general, test data and driver scripts are strongly coupled and need to be always synchronized [1]. If we look closer at the input data presented in Table 1 we can see that was designed only to accept two inputs and in order to support operations like 10 * 5 - 5 = 45 would require major changes on both test data and driver scripts.

The initial set-up effort of a data-driven testing framework is high [1]. Writing the right drivers need to be done by someone with programming skills, and not all of teams have resources to spend time on setting up testing frameworks. Although, the benefits gained will vastly outweigh this upfront cost when the test suit grows.

It is not appropriate for small systems where the cost of writing test cases is not so high [1]. If a data-driven approach is used it will entail more work, and will seem excessive. On the other hand, large and complex systems gain far more in saved effort than you put in.

3 Keyword-Driven Testing

Keyword-driven testing is a logical extension to data-driven testing [1]. Besides test data, it also makes use of keywords in order to instruct how the data is read from the external data source and to be interpreted by the test scripts. It takes the concept of data-driven testing even further [2] by adding keywords driving the test executing into the test data with the desire to be able to specify automated test cases without having to specify all the excessive detail. The test data file is expanded and it becomes a description of the test case using a set of keywords to indicate the tasks to be performed. Obviously, the driver script has to be able to interpret the keywords in order to execute the desired test case.

Back to the previous scenario on data-driven testing where the operation 10 * 5 - 5 = 45 would require major changes to both test data and driver scripts. The keyword-driven testing is the perfect technique to perform that kind of scenarios. They could be extended with no further development since the keywords are already implemented by the driver script. This way, the driver script is no longer tied to a particular feature of the software under test, nor indeed to a particular application or system [1].

This technique takes a descriptive approach to test case implementation since the test file describes the test case, by stating what the test case does and not how it does it. To implement an automated test case we have only to provide a description of the test case

more or less as we would do for a knowledgeable human tester. By using this approach we can build knowledge of the system under test into our test automation environment. People with business knowledge can concentrate on the test files while people with technical skills can concentrate on the driver scripts since it is possible to develop the test cases separately from the test scripts due to the abstraction created by the test files.

3.1 Test data

Test Case	Keyword	Value
Addition 01		
	Operand	5
	Operator	+
	Operand	5
	Result	10
Addition 02		
	Operand	5
	Operator	+
	Operand	10
	Operator	-
	Operand	1
	Result	14

Table 2: Keyword-driven test data with mathematical operations.

There is no real difference between handling keyword-driven and data-driven test data [1] [2].

One of the big decisions to make when designing a keyword-driven framework is the level of the keywords to be used [2]. When testing higher level functionality like business logic, low level keywords tend to make test cases very long, defined on Table 2, while higher level keywords, defined on Table 3 are much more usable. These higher level keywords shouldn't be implemented directly into the framework. Instead, they should be implemented using the same technique used for designing test cases. The main benefit in making it possible to create new keywords using the test design system is that they can be created and maintained easily without any programming skills. The drawback [2] from this approach is that the driver script will get more complex since it has to handle two kinds of keywords.

Test Case	Keyword	Value
Addition 01		
	Input	5
	Add	5
	Result	10
Addition 02		
	Input	5
	Add	10
	Subtract	1
	Result	14

Table 3: Keyword-driven test data with higher level keywords.

3.2 Processing test data

The driver script should be very similar to the one presented in Listing 1 and must be able to interpret the defined keywords and execute the matching action using the assigned values. Generally the number of scripts required for this approach is a function of the size of the software under test rather than the number of tests [1].

Listing 2: Example of keyword-driven driver script.

```
const TestData = require('./testData');
const Calculator = require('./calculator');

TestData.testCases().forEach(function(testCase) {
  testCase.keywords().forEach(function(keyword, value) {
    execute(keyword, value);
  });
});
```

In this keyword-driven approach there are a few elements that are crucial and must be implemented:

- **Data Parser**: Share the same responsibility as the data-driven parser defined on Figure 1, but has to handle multiple instructions per test case (Input, Add, Result).
- **Driver Script**: Share the same responsibility as the data-driven driver defined on Fig-

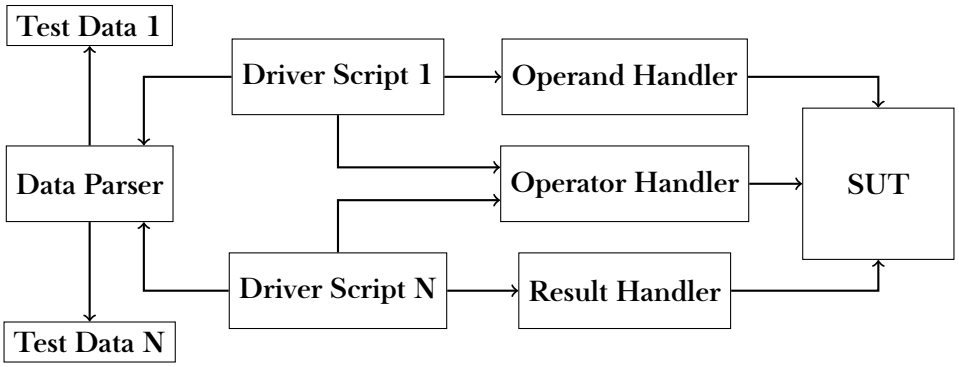

Figure 2: Keyword-driven system design approach.

ure 1, but instead of dealing directly with the test libraries, it needs to use specific handlers to translate the keywords and the data values that are read by the data parser in order to execute the tests.

- **Handlers**: Each action keyword corresponds to a single fragment of a test script (Operator, Operand, and Result handler), allowing the test execution tool to translate a sequence of keywords and data values into executable tests.

3.3 Advantages of keyword-driven testing

Keyword-driven testing has all the same benefits as data-driven testing since it shares the same principle of making easier to create new kinds of tests, even for non-programmers. But despite their resemblances, keyword-driven testing is a big step forward from pure data-driven testing where all tests are similar and creating new tests always require new code in the framework. Once the basic driver scripts are in place, it is possible to implement thousands of test cases with only a few hundred test scripts. This approach not only speeds up the implementation of automated tests, but also makes it possible to use testers without programming skills to implement them. The beauty of this approach is that it can be implemented in a tool independent way so that we can re-implement the driver scripts in another testing tool or programming language without loosing the investment in the test cases. All of the information about what to test is now contained in the data table, so the separation of test information from implementation is the greatest advantage in this technique.

3.4 Disadvantages of keyword-driven testing

The main problem with the keyword-driven approach is that test cases tend to get longer and more complex than when using the data-driven approach. If we look at the example from Table 1, the test case Addition 01 has only one line and it's very restrictive in terms of inputs. On the other hand, the same test case on Table 2 has four lines, is more complex to implement, but has lot more flexibility.

4 Model-Based Testing

Model-based testing is a software testing technique that uses test generation algorithms and a behavioral model of the system under test to generate test cases [5]. Unlike traditional development techniques which tend to focus on implementation, model-driven software development stresses the use of models at all levels of the software development process [3]. Like the data-driven and keyword-driven approaches to testing, model-based testing relies heavily on abstraction. Instead of writing hundreds of test cases the test designer writes an abstract model of the system under test, and then the model-based testing tool generates a set of test cases from that same model. It allow to easily generate a large test suite from the same model or regenerate the test suite each time the system requirements change. By following this approach, the test design time is reduced and several test suites can be generated by just using different test criteria. The purpose of model-based testing is to solve problems [5] that other testing processes do not fully address like:

- Automation of the design of functional test cases to reduce the design cost;
- Produce test suites with systematic coverage of the model;
- Reduction of the maintenance costs of the test suite;
- Automatic generation of the traceability matrix from requirements to test cases.

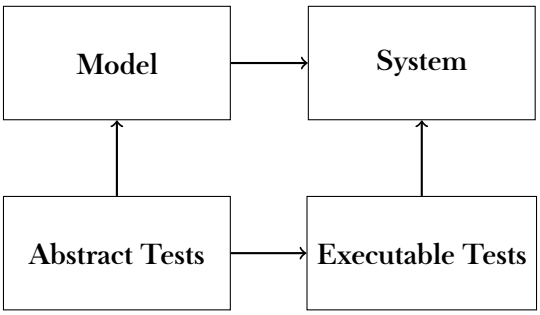

Figure 3: Model-driven definition

Summing up, the model is a partial description of the system under test, the abstract tests are derived from the model, and the executable tests are the concrete implementation of the abstract tests that can be run against the system under test.

4.1 How to model a system?

The first and most important step in modeling a system for testing is deciding the level of abstraction of the model itself. It should be clear which aspects of the system under test to include in your model and which aspects to omit. We should always have in mind that a smaller model is often more useful than a large and complex model, since it allow us to test each one of them independently and think about which operations should be included in the model before writing a model for the whole system.

When modeling a system is also important to think about the data that it manages, the operations that it performs, and the subsystems that it communicates with. The difficulty of test generation is usually highly dependent on the number and range of the input parameters, so applying the same abstraction principle to its input and output parameters helps to control the test generation effort.

After the test generation from the model and the execution of those tests against the system, each test that fails will point either to an error in the implementation of the system or to a mistake in the model. The value of model-based testing comes from the automated cross-checking between the model and the system implementation.

4.2 Challenges in Modeling

Model-based testing can improve considerably the test efficiency and test quality when the models are light and available. The elaboration of a model is, probably, the key for the success of model-based testing in practice, and if done wrongly [4] can lead to a set of undesirable outcomes:

- If complex models have to be completed before testing can start, this induces an unacceptable delay for the proper test executions;
- For complex SUT, like systems of systems, test models need to abstract from a large amount of detail, because otherwise the resulting test model would become unmanageable;
- The required skills for test engineers writing test models are significantly higher than for test engineers writing sequential test procedures.

4.3 Model-Based Testing and Agile methods

In the eyes of the agile practitioners, writing acceptance tests that specify what the system is supposed to do, brings lot of value:

- It forces the customer to specify precisely what is required;
- When the acceptance tests are âĂIJgreenâĂİ the customer has much more confidence that real and useful work has been done;
- An executable specification gives a clear and measurable objective to the development team.

Having the acceptance tests centered around a model it makes not only easier to develop a significant number of acceptance tests, but also to change the model and regenerate those same tests.

We can benefit even more from mixing model-based and acceptance tests, if the model itself is built around some agile principles:

- Test models should have a precise purpose;
- Test models should be light;
- Test models should grow incrementally through an iterative approach;
- Test models encourage discussion about the exact behavior of the system;
- Test models should not be used for documentation.

4.4 Prerequisites of model-based testing

It is important to have some high-level management support for the adoption of model-based testing, as well as some grassroots enthusiasm for model-based testing among the team who will put it into practice.

A testing team who has little experience with automated test execution might be wise to gain experience with an automated test execution approach like data or keyword-driven testing before using model-based testing, since that model-based testing tools implies that the test execution phase is already automated.

It is helpful to have some experience with designing interfaces and choosing good levels of abstraction, since the models for model-based testing are more precise than most UML models and require detailed modeling of the SUT behavior. Developing these kinds of models is more similar to programming than to designing UML use cases and class diagrams.

Model-based testing is most likely to be cost-effective when the execution of the generated tests can also be automated. If the SUT has to be tested manually because the test

steps require human interaction, then the cost of that manual testing may dominate any cost savings of model-based testing.

4.5 Advantages of model-based testing

Model-based testing can lead to less time and effort spent on testing if the time needed to write and maintain the model plus the time spent on directing the test generation is less than the cost of manually designing and maintaining a test suite. The use of an automated test case generator based on algorithms and heuristics to choose the test cases from the model makes the design process systematic and repeatable.

4.6 Disadvantages of model-based testing

One of the fundamental limitations of model-based testing is that it cannot guarantee to find all the differences between the model and the implementation, not even if there are hundreds of test cases being generated by the system model.

Model-based testing is not as trivial as other testing techniques, and requires a steep learning curve and different testing skills: modeling and programming skills. It is desirable to have a reasonably mature testing process and some experience with automated test execution before fully adopt this approach.

Up to date requirements plays a big role in model-based testing. The test cases are very coupled to the requirements, and requirements change, often. If the model is build based on outdated requirements, it will lead to unreliable results.

When one of these tests fails, it's hard to check if the failure is caused by the system under test, the driver, or even from the model itself. This process makes more difficult and time-consuming to find the root cause of the failing test.

Since this approach can generate huge numbers of tests, it becomes necessary to move toward other measurements of test progress instead of relying on the number-of-tests metric. Business code, requirements and model coverage are some of the alternatives.

5 Hybrid-Driven

6 Conclusion

References

[1] Mark Fewster and Dorothy Graham. *Software Test Automation: Effective Use of Test Execution Tools*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[2] Pekka Laukkanen and Pekka Laukkanen. Data-driven and keyword-driven test automation frameworks, 2007.

[3] M. Mussa, S. Ouchani, W. A. Sammane, and A. Hamou-Lhadj. A survey of model-driven testing techniques. In *2009 Ninth International Conference on Quality Software*, pages 167–172, Aug 2009.

[4] Jan Peleska. Industrial-strength model-based testing - state of the art and current challenges. *Electronic Proceedings in Theoretical Computer Science*, page 3âĂŞ28, 2013.

[5] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.