ISSN 1407-7493

COMPUTER SCIENCE
DATORZINĀTNE

2008-34

APPLIED COMPUTER SYSTEMS
LIETIŠĶĀS DATORSISTĒMAS

AUTOMATED TEST STATE MANAGEMENT FRAMEWORK

Aleksandr Sukhorukov, *Riga Technical University,*
Meza 1/3, Riga, LV 1048, Latvia, assistant, M.sc.nat,
Aleksandrs.Suhorukovs@cs.rtu.lv

Larisa Zaitseva, *Riga Technical University,*
Meza 1/3, Riga, LV 1048, Latvia, professor, Dr.sc.ing.,
Larisa.Zaiceva@cs.rtu.lv

Software engineering, Software testing, Test automation

1. Introduction

Automated software testing is becoming a vital area in software engineering. Modern tools for functional test automation have evolved significantly in terms of supporting various graphical user interface (GUI) environments [1]. Frameworks for unit test automation have matured and are present in variety of forms [2].

One of the most important problems of test automation regardless of testing level is architecture and design of test script suites. When amount of tests is large – hundreds and thousands – code duplication and lack of common design will make test scripts non-maintainable [1].

In this article we present a concept of framework that solves problems of implementation complexity and execution speed of automated tests. The second section of the article describes structure of automated test; the third section introduces the problem and some partial solutions; the fourth section presents more complete solution – Test State Management Framework.

2. Automated test structure

Automated test typically consists of several parts. In unit testing there is a common approach to isolate three distinct parts of test [3]:

- Setup – the first part, that performs non-testing tasks in order to assure pre-conditions of actual test are met;
- Body – actual testing occurs here. This part assumes that pre-conditions are met already, performs necessary steps and verifies correctness of application's behaviour;

- Teardown – the last part where application is recovered from test's consequences, like freeing resources, putting application in some base state etc.

The same principle can be applied to any automated test because regardless of its actual composition, conceptually it can always be divided into these three distinct parts.

Body part is the substantial part of automated test. It implements the actual test case and usually it should be kept as simple as possible in order to be easy to understand. This rule is not always obeyed in real life, however one Body part's feature usually is true – it does only those things that are necessary according to test's design.

Setup and Teardown parts are auxiliary parts of automated tests. They are not required by test's purpose, but rather serve as means for helping Body to achieve its goal.

3. Managing complexity of auxiliary parts

In this part we overview the problem of auxiliary parts of automated tests and look for several approaches to their management.

3.1. Why auxiliary parts are complex

In complex systems auxiliary parts can be more complex than Body. In unit tests this problem usually is not so pronounced as in system level functional tests. For example in object-oriented paradigm, unit test focuses on a single class [4]. So, Setup usually consists of constructing appropriate object, filling it with necessary data and putting it into necessary state, but Teardown destroys the object and frees memory. This usually takes just a few lines of code that do not look huge if compared with Body. However, auxiliary parts can be very complex even in unit testing.

In system level tests problem of complexity of auxiliary parts is more important. There can be many complex tasks that Setup and Teardown can perform. Especially Setup is usually overloaded with tasks [5]. It can

- perform several SQL queries to assure necessary state of database,
- create some files on disk,
- change some configuration either by modifying configuration file or even do it in user interface,
- open application and drive it to necessary window or page.

Teardown usually is simpler, but also can do many complex things:

- modify or delete some database records, or even rollback whole database from backup,
- delete some files or replace modified files with some standard versions,
- return application to main window or close it at all.

And all these tasks can be necessary just for a very short tests consisting of a few simple steps. When Setup and Teardown parts of automated test script are several times longer and more complex than actual Body, test automators may feel like they perform inefficiently. This may lead to oversimplifying auxiliary parts making tests less reliable or to overloading Body

with several conceptually different test cases making tests less understandable and manageable.

3.2. Independence of tests

The main reason why auxiliary parts may grow large is assuring independence of different tests. There are several advantages of making tests independent of each other [3]:

- Testers are free to create test suites where they can combine any test cases in any order;
- If test does not depend on other test's execution results, it is more reliable as there are no unexpected side-effects;
- Tests are more manageable – creating new tests or changing existing ones does not affect other tests.

If this principle is not obeyed and tests are dependent on each other, these positive effects can be lost (test cases must be executed in predefined order, maintenance of tests becomes more difficult etc). If amount of test cases is large, such approach can negatively affect maintenance and usability of tests. Therefore we will assume here that test independence concept should be used in architecture of automated tests.

3.3. Basic test flow

Test independence is assured by concept of Base state of the system under test. There is one single base state where executions of all tests begin. As actual test Body have its own unique preconditions, Setup part should change application's state from Base state to the state where Body's preconditions are met. We will call it Start state. When Body execution finishes, application is in state which we will call End state, where test's post-conditions are met. Teardown part's task then is to return application from End state back to Base state. This process is visualized in Figure 1.

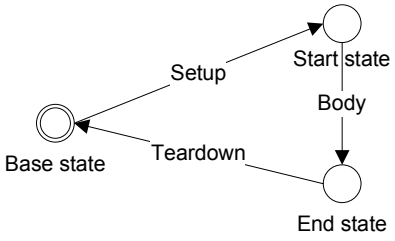

Figure 1. Basic test flow

This scheme is very simple and straightforward. There are two problems with it however:

1) Complexity of implementation. Each test has its own Setup and Teardown that should be implemented separately. Implementation and maintenance of auxiliary parts can take more effort than implementation and maintenance of Body.
2) Speed of execution. When tests are executed each test's Setup and Teardown are executed. These parts can execute much longer than Body. In large test suites this can be very inefficient, as most of the time is spent on auxiliary parts execution. Especially this is true in system-level tests working on GUI level, where execution is rather slow [5].

These two problems should be addressed somehow. Let's look at several commonly used patterns for making tests more efficient.

3.4. Shared auxiliary parts

Sometimes Start and End states of a test coincide. This occurs when Body does not change anything in application or changes made are not relevant to consider Start and End states as different. In this case Setup and Teardown can be shared (e.g. Figure 2) by several test cases making automated test development faster. In order to make execution faster as well it is possible to make Setup run once then execute Bodies of the tests and then run Teardown to conclude execution. Depending on implementation this can break independence of tests in this group (if sequence Setup → Body1 → Body2 → … → BodyN → Teardown is hard-coded) or require additional features of framework (it should determine automatically that if such tests are run one after another, Setup and Teardown should be run just once).

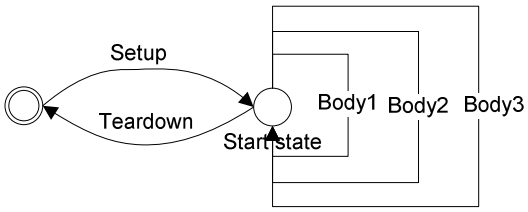

Figure 2. Shared Setup and Teardown

This approach is usual for unit testing frameworks like jUnit, where single Setup and Teardown procedures are shared by several test cases in test suite, and can be also successfully used in system-level test automation. It optimizes both implementation and execution time as Setup and Teardown are implemented and executed once for several tests group.

The only drawback is that usually each test has some side-effects. And in order to make Start and End states coincide some steps, that are more appropriate for Setup or Teardown, are placed into Body implementation. This seems to break the concept but in fact it just makes it more complex. This complicated Body actually can be considered as having Setup, Body and Teardown parts itself. Now for each test instead of having single Setup we have one general Setup that is common to several tests and one specific Setup. The same is with Teardown. This new complexity is illustrated in Figure 3.

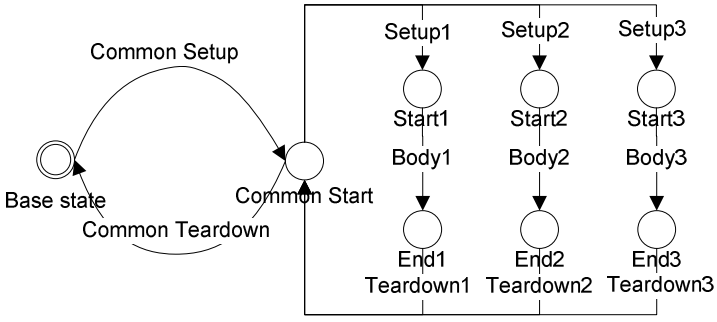

Figure 3. Doubling auxiliary parts

This approach optimizes basic test flow scheme, however its use is limited to tests having the same Start state. In complex systems there can be a large amount of such groups each group having their own Common Setup and Common Teardown parts that can still be complex and slowly executing.

3.5. Nested test suites

Commonly known unit testing frameworks allow test suites consist of other test suites [3]. Thus Setup and Teardown of higher level suite envelop Setup and Teardown of lower level suites, allowing multi-level Setups and Teardowns like. In Figure 4 test being executed is included into four level suite hierarchy each having its own Setup and Teardown sections.

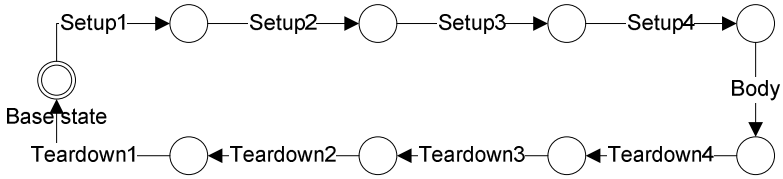

Figure 4. Test in multi-level suite

This approach breaks down Setup and Teardown sections in reusable parts. However these frameworks usually don't allow executing single test. In order to execute test included in schema shown at Figure 4, the highest-level suite should be run completely, and it could contain thousands of tests. This approach could be acceptable and even advisable in unit testing, where tests execute quickly and should be executed repeatedly often. This can be inefficient for tests working on GUI level which are much slower in execution and such a big test suite can take hours to complete [5].

3.6. Compound auxiliary parts

Because of mentioned problems for system-level tests working on GUI level the simplest design is used usually as illustrated in Figure 1 [6]. It is easy to understand and easy to manage. If some parts of Setup or Teardown are common to many tests (like opening necessary window, cleaning the database etc), these steps are usually implemented in separate functions and called from Setup or Teardown parts of tests that need them. Isolating these common functions help eliminating duplication in test code and keep design clean. So, test illustrated in Figure 4 could be implemented as (in pseudocode):

```
Test {
    Setup {
        Setup1; Setup2; Setup3; Setup4;
    }
    Body {
        <Body actions>
    }
    Teardown {
        Teardown4; Teardown3; Teardown2; Teardown1;
    }
}
```

SetupN and TeardownN functions can be easily shared between different tests, so, the problem of complexity of implementation is solved, as Setup and Teardown can be built by using existing blocks. This approach is commonly used nowadays [3].

One problem remains however – tests are still tied to Base state, so speed of execution still is not optimal. Setup must go all the way from Base state to Start state, and Teardown must go back fully. Even if Start state of second test can be very easily achieved from End state of the first test, state transition will go through Base state.

4. Solution – Test State Management Framework

In this part we present a solution that addresses both problems – complexity of implementation and speed of execution. The solution is a framework that executes automated tests of special design in optimal way. We will call it Test State Management Framework (or simply the Framework).

4.1. Some observations

In different conditions, like running test separately from others or in some test suite, different ways of assuring Start state may be more optimal. So, there is a difference whether

1) the test case is the first (or the only) one in test suite or
2) it will execute after some other test case.

In the first case Setup part must traverse full path from Base state to Start state. In the second case, it would be more optimal if Setup would go the shortest way moving from End state of previous test case to Start state of new test. The same is with Teardown. If test case is not the last one in test suite, Teardown should better go the shortest way moving from test case's End state to next test case's Start state. So, Teardown of previous test case becomes Setup of next test case. We will call this combined Teardown/Setup procedure Transition. Example of two test cases is shown in Figure 5.

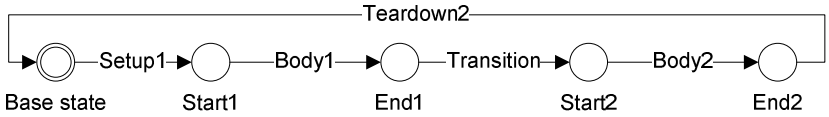

Figure 5. Transition

Sometimes it even may appear that Start state of some test case can coincide with End state of some other test. For example, if in Figure 5 states End2 and Start1 coincide then execution sequence selected there is not optimal. If we execute these test cases in reverse sequence we will not need any transition between them (like in Figure 6).

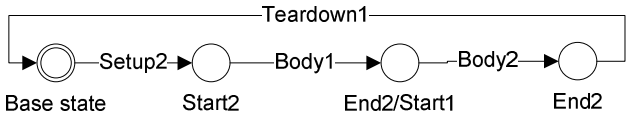

Figure 6. Without transition

4.2. Features needed

Our observations lead to realizing necessity of test execution framework having useful features:

1) Setup and Teardown should not be parts of test itself, the framework should somehow generate them from available Transitions depending on sequence of test cases that must be executed;
2) As one sequence of tests can execute more efficiently than the other sequence, framework should figure out the most optimal sequence and execute this sequence.

In order to perform these tasks, framework will have to have some additional information supplied with test scripts. Now we will examine how automated tests should be designed for such framework.

4.3. Test design

In Test State Management Framework there are two types of automated scripts:

1) Test scripts, which perform actual testing;
2) Auxiliary scripts, which are needed just as Transitions between two states. Auxiliary scripts usually should not contain any verifications as their purpose is different.

Setup and Teardown parts completely disappear from tests. Automated script consists of two parts:

1) Script Body implemented as executable code that drives the application and verifies correctness.
2) Script Metadata including:
    a. Name of Start state where necessary pre-conditions are met;
    b. Name of End state where script's Body finishes its execution;
    c. Estimated time of execution of Body.

Regardless of script's type Metadata should be included in script's description.

All scripts are contained in single repository, where framework can access them and execute. Test suites are constructed as sets of Test scripts (scripts of type 1) that must be executed.

This information is everything that is necessary to create automated tests for the Framework. The Framework will do all other job, warning if some additional Transition scripts are needed for execution.

4.4. Framework job

When the Framework is given a test suite that must be executed, it performs several tasks:

1) Extracts Metadata from each script in repository. These data for single script can be illustrated as in Figure 7.

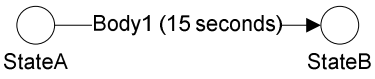

Figure 7. Script Metadata

2) Constructs graph in memory, perceiving each script's Metadata as a single arc (like in Figure 8). Both Test and Auxiliary types of scripts are included in the graph as both can serve as transitions between states. Auxiliary scripts are necessary to make the graph connected and all states achievable if Test scripts are not enough to assure this feature.

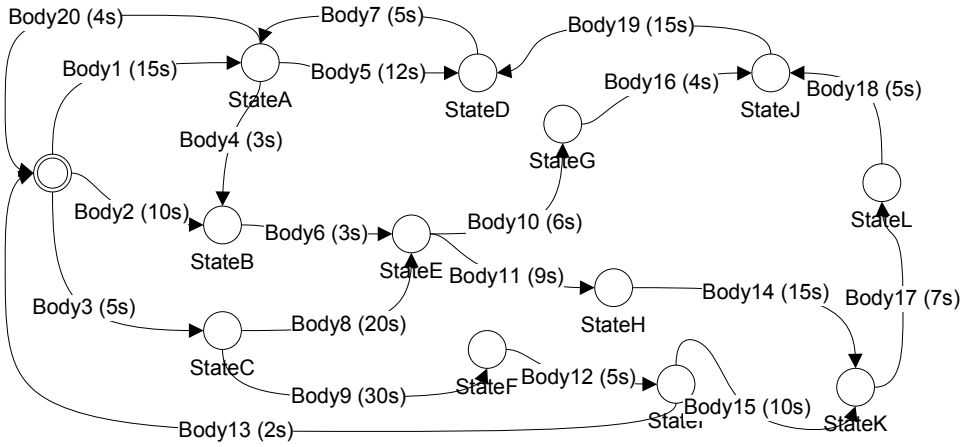

Figure 8. State transition graph

3) Calculates shortest path (taking estimated execution time as a weight of arc), necessary to execute scripts in given test suite. So, some scripts (both Test and Auxiliary) not included in test suite may be included in the path in order to achieve necessary states. The Framework gives a warning if graph is disconnected, so, some necessary state cannot be achieved.
4) The Framework executes this path.

So, using such framework gives an advantage that only needed parts are executed and optimal state transitions are assured. On the other hand, any tests (implemented as Test scripts) from repository can be included in test suite, and tester has not to worry about selecting the most optimal sequence of tests, as the framework does this automatically. As shows authors' experience GUI-level test (especially in web environment) execution is usually slow and may take minutes to complete, but shortest path discovery for graphs containing thousands of states takes few seconds or less on modern computers, usage of the Framework can significantly reduce total time of execution. For quantitative measures of the improvement additional research is required addressing different types of systems and tests of different complexity.

4.5. Implementation issues

In order to ensure necessary functionality the Framework should contain the following modules:

1) Repository that will store and manage all scripts and provide means for adding new scripts and editing existing ones.
2) Test suite management module should provide means for selecting tests from test repository in order to combine them in single unit of execution. It should allow saving created test suites as well as editing existing ones.
3) Execution path generation module should construct graph from scripts' Metadata and calculate the fastest sequence of scripts that starts and finishes in Base state and traverses all arcs that correspond to tests contained in selected test suite. This problem is equivalent to discovery of shortest path in weighted oriented graph through selected arcs.
4) Execution modules should provide means to actually execute scripted actions on application under test. As different types of GUI (like web, java, standard Windows

etc) or API have very different mechanisms of their automation, there can be several execution modules – each for its own GUI or API type.

5) Results management module should store results of test suite execution and allow comparison of results of different executions.

Mentioned modules are the core ones, additional modules could include user rights management, scheduling test execution, features for storage of additional information, etc.

5. Conclusion

Proposed Test State Management Framework could help to implement and execute automated tests for complex systems where problem of test code amount and its maintenance. The first prototype of the Framework is going to be implemented this year to help organize automated testing of e-learning system developed in Riga Technical University and serve as base automated test management tool in some projects of JTC Ltd.

Described model of test state management can be further extended with parameterized states and test bodies. Then along with test and state names, parameters are provided that affect behaviour of Bodies. This approach is useful if several test cases share the same test procedure [7]. Framework can accomplish additional tasks, like deducting parameters that should be given to intermediate Bodies in order to achieve necessary state with appropriate parameters. Parameterized Test State Management Framework, its features and implementation issues can be further investigated.

References

1. Kaner, C. (2000). Architectures of Test Automation. Software Testing, Analysis & Review Conference (Star) West. San Jose, CA.
2. Unit testing software / Internet. - http://www.xprogramming.com/software.htm (Visited on 1.Oct.2007)
3. Hunt, A., Thomas, D. (2003). Pragmatic Unit Testing in Java with JUnit. The Progmatic Programmers, USA.
4. McGregor, J., Sykes, D. (2001). A Practical Guide to Testing Object-oriented Software, Chapter 5. Addison-Wesley, USA.
5. Cohen, F. (2004). Java Testing and Design: From Unit Testing to Automated Web Tests, Chapter 8. Prentice Hall PTR, New Jersey, USA.
6. Li, K., Wu, M. (2005). Effective GUI Test Automation: Developing an Automated GUI Testing Tool, Chapter 2. SYBEX, Alameda, CA.
7. IEEE Computer Society (1998). IEEE Standard for Software Test Documentation, IEEE Inc, USA.

Suhorukovs A., Zaiceva L. Automatizēto testu stāvokļu pārvaldības karkass

Raksts ir veltīts automatizēto testu projektēšanas un implementēšanas problēmām, proti, testēšanas vadībai ar mērķi samazināt realizācijas sarežģītību un palielināt izpildes ātrumu. Ir analizēta šo divu problēmu būtība un rašanas cēloņi automatizētas testēšanas kontekstā. Ir aprakstīta automatizētās testēšanas struktūra, kas ietver pamatdaļu un divas palīgdaļas, kā arī izklāstītas dažādas pieejas testēšanas organizēšanai, kuras atspoguļotas ar stāvokļu pāreju modeļiem. Kā iespējamais abu problēmu risinājums (realizācijas sarežģītība un izpildes ātrums) ir piedāvāts testu stāvokļu pārvaldības karkass. Galvenā karkasa ideja ir tāda, ka testi var tikt izmantoti sākuma stāvokļa nodrošināšanai citiem testiem, un līdz ar to pareizas testu izpildes secības izvēle var

ievērojami palielināt izpildes ātrumu. Šo izvēli var automātiski veikt piedāvātais karkass, izmantojot automatizēto testu metadatus – informāciju par testa sākuma un beigu stāvokļiem, kā arī prognozējamu testa izpildes laiku. Kā pozitīvs blakusefekts, stāvokļu pārejas galvenokārt tiek nodrošinātas ar citu testu palīdzību nevis ar atsevišķām testu palīgdaļām, tādējādi samazinot realizācijas sarežģītību. Ir apraksta pieeja šāda karkasa projektēšanai un realizācijai, kā arī apskatīti daži tā pielietošanas aspekti sarežģīto sistēmu testēšanai. Principi, uz kuriem balstās piedāvātais karkass, tika veiksmīgi izmantoti trijos SIA JTC testēšanas automatizācijas projektos, kur tie parādīja savu augsto efektivitāti.

Sukhorukov A., Zaitseva L. Automated test state management framework

The paper studies problems of automated testing design and implementation, especially, of testing management on purpose to decrease a complexity of implementation and to increase a speed of execution. Essence and causes of these two problems are analyzed in the context of automated testing. Automated test structure that includes the basic part (Body) and two auxiliary parts (Setup and Teardown) is outlined. Different approaches of managing complexity of auxiliary parts are described using models of state transitions during testing. As a possible solution for both of these problems (complexity of implementation and speed of execution) a test state management framework is offered. Idea behind this framework is that tests can be used for assuring start-state for other tests, and therefore selecting appropriate sequence of execution can significantly improve overall speed of execution. This selection can be done automatically by the offered framework using metadata of automated test – information about start and end state of test as well as estimated execution time of test. As a positive side effect, other tests are mostly used as means for state transition instead of separate auxiliary parts of test, thus reducing complexity of implementation. The paper presents an approach to design and implementation of such framework and addresses several issues of its application to testing complex systems. Principles on which proposed framework is based were successfully applied in three JTC Ltd testing automation projects where they showed their high efficiency.

Сухоруков А., Зайцева Л. Каркас управления состояниями автоматизированных тестов

Статья посвящена проблемам проектирования и реализации автоматизированных тестов, а именно, управлению тестированием с целью сокращения сложности реализации и повышения скорости выполнения. Сущность и причины этих двух проблем анализируются в контексте автоматизированного тестирования. Описана структура автоматизированного тестирования, включающая основную и две служебных части. Рассмотрены различные подходы к организации тестирования, представленные также в виде моделей переходов состояний. Как возможное решение обеих проблем тестирования (сложности реализации и скорости выполнения), предложен каркас управления состояниями тестов. Идея каркаса состоит в том, что тесты могут быть использованы для обеспечения начального состояния других тестов, и поэтому правильный выбор последовательности выполнения может значительно повысить общую скорость выполнения. Этот выбор может автоматически выполнять предлагаемый каркас, используя метаданные автоматизированного теста – информацию о начальном и конечном состоянии теста, а также прогнозируемое время выполнения теста. Как положительный побочный эффект, для переходов между состояниями используются в основном другие тесты вместо отдельных служебных частей тестов, а это приводит к сокращению сложности реализации. Описан подход к проектированию и реализации такого каркаса, а также некоторые аспекты его применения для тестирования сложных систем. Принципы, на которых основывается предложенный каркас, были успешно использованы в трех проектах SIA JTC, где они показали свою высокую эффективность.