December 2014

testing experience

The Magazine for Professional Testers

The Three Pillars of Agile Quality and Testing
by Robert Galen

Testing the Internet of Things – The Future is Here
by Venkat Ramesh Atigadda

... and many more

printed in Germany
print version 8,00 €
free digital version
www.testingexperience.com
ISSN 1866-5705

Missed Part I?
Read it in
issue No. 27!

By Rudolf de Schipper & Abdelkrim Boujraf

Test-Driven Developments are Inefficient; Behavior-Driven Developments are a Beacon of Hope?

The StratEx Experience (A Public-Private SaaS and On-Premises Application) – Part II

In part 1 of this article, we outlined a number of criticisms and roadblocks that we have encountered while looking for the best way to test our application.

We know it is easy to criticize, but this was not for the sake of being negative. We believe that our experience has led to some valuable insights, apart from the points we simply do not like. Further, we believe our experiences are not unique. So, in this second article we want to take a look at what can be done to test our software efficiently.

What we describe here is not all implemented today. As we said in the previous article, this is part of a journey, a search for the optimum way of doing things. You might wonder why? There are a couple of reasons for this. First and foremost, as a small startup company, our resources are limited. And, as Google says, "scarcity brings clarity" [1], so we have to be careful what we spend our time and energy on. Second, when doing our research on testing methodologies and how to apply these best, there was one recurring theme: there is never enough time, money, or resources to test. This can probably be translated into "management does not give me enough, because they apparently believe that what they spend on testing is already enough". Now here comes the big question: what if management is right? Can we honestly say that every test, every check, every test department even, is super-efficient? We may argue that test effort may reach up to x % of development effort (30 % has been given as a rough guideline). Well then, if by magic, development effort is sliced down to one fifth, is it not logical to assume that the test effort should be reduced by the same factor? And how would this be achieved? We want to explore this here.

This is where we came from. We generate large parts of our code. This reduces development time dramatically. For a small startup this is a good thing. But this also means that we must be able to reduce our testing time. And that was the reason we had a good and hard look at current testing methods, what to test, when to test, and how to test.

Testing a Web Application Deployable as Public-Private Cloud and On-Premises Software

First, let's frame our discussion. The application we are developing is rather standard from a testing point of view: web-based, multi-tier, a back-end database, and running in the cloud. The GUI is moderately sprinkled with JavaScript (JQuery [2] and proprietary scripts from a set of commercial off-the-shelf (COTS) UI controls like DevExpress [3]and Aspose [4]). The main way to interact is through a series of CRUD [5] screens. We can safely say that there are probably thousands of applications like this, except that the same piece of code is deployable as a Private Cloud and Public Cloud application, as well as on-premises software. This is our target audience for this article. We are not pretending to describe how to test military-spec applications or embedded systems, for example.

What do we want to achieve with our tests? In simple terms, we are not looking for mathematical correctness proofs of our code. Nor are we looking for exhaustively tested use cases. We want to be *reasonably certain* that the code we deploy is stable and behaves as expected. We are ready to accept the odd outlier case that gives an issue. We believe our bug-fixing process is quick enough to address such issues in an acceptable timeframe (between 4 and 48 hours).

Let's look a bit closer at the various types of tests we might need to devise to achieve such reasonable assurance.

Testing a CRUD Application

The application presents a number of screens to the user, starting from the home screen, with a menu. The menu gives access to screens, mostly CRUD-type, while some screens are process-oriented or serve a specific purpose (a screen to run reporting, for example).

A CRUD screen has five distinct, physical views:

1. The items list (index), i.e., the Contracts list
2. The item details, i.e., the Work Page details
3. The item edition, i.e., edit the Activities details
4. The item creation, i.e., create a new Type of Risk
5. And the item deleting, i.e., delete a Project and its related items

Possible actions on each screen are standardized, with the exception of the details screen, where specific, business-oriented actions can be added. You may think of an action such as closing a record, which involves a number of things such as checking validity, status change, updating log information, and maybe creating some other record in the system. In essence, these actions are always non-trivial.

Testing a Generated vs. a Hand-Coded Piece of Software

All CRUD screens are made of fully generated code, with the exception of the business actions, which are always hand-coded.

The non-CRUD screens are not generated and always hand-coded. Needless to say we try to keep the number of these screens low.

We have observed that the generated code and screens are usually of acceptable initial quality. This is because the number of human/manual activities to produce such a screen is very low. The code templates that are used by the generator obviously have taken their time to be developed. This was however a localized effort, because we could concentrate on one specific use case. Once it worked and had been tested (manually!), we could replicate this with minimal effort to the other screens (through generation). We knew in advance that all the features we had developed would work on the other screens as well. An interesting side-effect of this method is that if there is an error in the generated code, the probability of finding this error is actually very high, because the code generation process *multiplies* the error to all screens, meaning it is likely to be found very quickly.

The hand-code screens are on the other side of the scale. They present a high likelihood of errors, and we have also found that these screens are prone to non-standard look and feel and non-standard behavior within the application. When compared to the approach of generating highly standardized code, the reasons for this are obvious.

Testing Business Actions

The business actions are the third main concern for testing. These are non-trivial actions, highly context (data, user) dependent, and with a multitude of possible outcomes. We have not yet figured out how to test these artifacts automatically due to the amount of cases we would need to take into account. Each change in our logic needs a complete refactoring of those tests that will certainly produce most of the complaints from of our beloved customers.

Testing the User Interface Using BDD

A final concern is the UI testing. Even with highly standardized screens, we value this type of testing, for three reasons:

- First it is the way to run a partial end-to-end test [6] on your screens as we test partially the content of the database after a screen is tested.
- Second, it is what the user actually sees. If something is wrong there, there is no escape.
- Third, we like to use such test runs to record documentation, and demo and training videos using Castro [7], Selenium [8], and Behave [9] – mostly open source software.

We believe that this context is relatively common, with the possible exception of massive code generation and the use of tests to document the application (and we would recommend these as something to consider for your next project!), so it makes sense to examine how these different areas can be tested efficiently.

For the generated CRUD screens, tests should be standardized and generated. Do we need to test all the generated screens? Given the fact that the tests are generated (so there is no effort involved in creating the tests), we would say that you must at least have the possibility of testing all screens. Whether you test them for every deployment is a question of judgment.

Hand-coded screens, if you have them, probably require hand-coded tests. Yet, if you have information available that allows you to generate (parts of) your tests, do it. It reduces the maintenance cycle of your tests, which means you improve the long-term chances of survival of your tests. We have not found convincing evidence to state that hand-coded (non-standardized) screens can be fully described (see Table 1) by a set of BDD/Gherkin tests or briefly described (see Table 2). The simple fact is that it would require a large amount of BDD-like tests to fully describe such screens. One practice we have observed is to have one big test for a complete screen; however, we found that such tests quickly become complex and difficult to maintain for many reasons:

1. You do not want to disclose too much technical information to the business user, e.g., username/password, the acceptable data types, the URL of the servers supporting the factory acceptance test (FAT), system acceptance tests [10] (SAT) and the live application.
2. You need to multiply the number of features by the number of languages your system supports.
3. You want to keep the BDD test separate from the code that the tester writes, as the tests depend on the software architecture (Cloud, on-premises) and the device that may support the application (desktop, mobile).
4. A database containing acceptable data might be used by either the business user or the tester. The data might be digested by the system testing the application and reduce the complexity of the BDD-tests while increasing the source code to test the application

```
1.  # file: ./Create_Contract_Request_for_offer.feature
2.  Feature: Create a Request for offer
3.      As a registered user,
4.      I want to create a Request for offer for a project
5.  Background:
6.      Given I open StratEx "<url>"
7.      When I sign up as "<username>"
8.      Then I should be signed in as "<user_first_last_name>"
9.  Scenario Outline:
10.     Then I click on "Contract" menu item
11.     Then I click on "Request for offer" menu item
12.     Then I click on "Create new" menu item
13.     Then I select "<project_name>" from the "Project"
        dropdown
14.     Then I set the "Title" box with "<project_title>"
15.     Then I click on "Save" button
16.     Then I check that the "Project" field equals
        "<project_name>"
17.     Then I check that the "Title" field equals
        "<project_title>"
18.     And I click on the link "Logout"
19. Examples: staging
20.     | url                                    | username
        | user_first_last_name    | project_name   | project_
        title |
21.     | https://staging.<your application>.com | a Username
        | a firstname, a lastname | a project name | a project
        title |
```

Table 1. BDD Definition (Full Description): Create a Request for Offer "11_Create_Contract_Request_for_offer.feature"

```
# file: ./Create_Contract_Request_for_offer.feature
Feature: Create a Request for offer
    As a registered user,
    I want to create a Request for offer for a project
Background:
    Given I open StratEx "<url>"
    When I sign up as "<username>"
    Then I should be signed in as "<user_first_last_name>"
Scenario Outline:
    Then I create one Request for offer
    And I click on the link "Logout"
Examples: staging
    | url                                    | username   |
    user_first_last_name    |
    | https://staging.<your application>.com | a Username |
    a firstname, a lastname |
```

Table 2. BDD Definition (Brief Description): Create a Request for Offer "11_Create_Contract_Request_for_offer.feature"

```
@then(u'I create one Request for offer')
def step_impl(context):
    # click | 'Contract' menu item
    context.browser.find_element_by_xpath(
        "//a[contains(text(),'Contract')]").click()
    context.browser.dramatic_pause(seconds=1)
    # click | 'Request for Offer' menu item
    context.browser.find_element_by_xpath(
        "//a[contains(text(),'Request for Offer')]").click()
    context.browser.dramatic_pause(seconds=1)
    # click | 'Create New' menu item
    context.browser.find_element_by_xpath(
        "//a[contains(text(),'Create New')]").click()
    context.browser.dramatic_pause(seconds=2)
    # select | id=Project | label=StratEx Demo
    Select(context.browser.find_element_by_id("Project")).
        select_by_visible_text(context.testData.find(
        ".//project_name").text)
    context.browser.dramatic_pause(seconds=1)
    # type | id=Title | Horizon 2020 dedicated SME
        Instrument - Phase 2 2014
    context.browser.find_element_by_id("Title").clear()
```

Table 3. Excerpt from "11_Create_Contract_Request_for_offer.py"

At StratEx, we our current practice is to write brief BDD, after many attempts to find a right balance between writing code and BDD-tests. We did choose the Python Programming language (see Table 3) to implement the tests, because Python [11] is readable even by business users and can be deployed on all our diverse systems made of Linux and Windows machines.

Business actions are hand-coded too, but such actions are good candidates for BDD-tests, described in Gherkin. As we mentioned before, Gherkin is powerful for describing functional behavior, and this is exactly what the business actions implement. So there seems to be a natural fit between business actions and BDD/Gherkin. The context can usually be described in the same way as for UI tests (see above). Can such tests be generated? We believe that the effort for this might outweigh the benefits. Still, using Gherkin to describe the intended business action and then implementing tests for it seems like a promising approach.

Conclusion

This broadly covers the area of functional testing, including UI tests. The question obviously occurs as to what else needs to be tested, because it is clear that the tests we describe here are not ideal candidates for development-level (unit) tests – they would simply be too slow. In various configurations, the tests we described above would run in a pre-deployment scenario, with more or less coverage: run all screens, run only the hand-coded screen tests, run some actions as smoke tests, etc.

We believe that the most relevant tests for the development cycle are the ones related to the work the developer does, i.e., producing code. This means that generated code can be excluded in principle (although there is nothing against generating such tests). It focuses therefore on hand-coded screens and business action implementation.

Starting with the business action implementation, we observe that this only requires coding in non-UI parts of the application: the model [12] and the database. It has been shown that it is possible to run unit-like tests against the model code and against the controller code. Unit tests against the model can be used to validate the individual functions (as in "normal" unit tests), while tests against the controller will actually validate that the action when invoked by the user from the UI will yield the expected result. In that sense, this kind of test runs like a UI test without the overhead (and performance penalty) of a browser.

What is so special about this approach? First, these are not real unit tests because they do not isolate components. They test an architectural layer, including the layers below it. This means that, when testing a model, the database will be accessed as well. This is a deliberate trade-off between the work required to make components "unit-testable" and the impact of testing them "in context". It means we have to consider issues such as database content and we need to accept that these tests run slower than "real" unit tests. However, because we have far fewer of this type of test (we only implement the tests for the business actions, which is between two and ten per screen), the number of these tests will be around 100–200 for the complete application. We believe that this is a workable situation, as it allows developing without having to consider the intricacies of emulated data, such as mocks or other artificial architectural artifacts, to allow for out-of-context testing. In other words, we can concentrate on the business problems we need to solve.

An additional benefit here is that this allows us to test the database along with the code. Database testing is an area we have not seen covered often, for reasons that somewhat elude us.

In summary, we have presented here a method for efficiently testing large parts of web-based software by using elements of code generation to generate automatable tests, and by using BDD concepts to model tests for non-generated screens and non-generated business actions. Further, we have described a method for context-based unit testing that, when combined with generated code and tests, yields

an acceptable trade-off between development efficiency and time spent on testing.

This article has not covered other areas of testing, such as performance and security tests. Currently StratEx has no immediate concerns in these areas that required us to critically observe how we validate the application in this respect. ■

References

[1] "How Google tests software" (Whittaker, Arbon et al.)

[2] jQuery: *http://jquery.com*

[3] DevExpress: *https://www.devexpress.com/*

[4] Aspose: *http://www.aspose.com*

[5] Create, Read, Update, Delete

[6] End-to-end test: *http://www.techopedia.com/definition/7035/end-to-end-test*

[7] Castro is a library for recording automated screencasts via a simple API: *https://pypi.python.org/pypi/castro*

[8] Automating web applications for testing purposes: *http://www.seleniumhq.org*

[9] Behave is behavior-driven development, Python style: *https://pypi.python.org/pypi/behave*

[10] System Acceptance Test: *https://computing.cells.es/services/controls/sat*

[11] Python Programming Language: *https://www.python.org/*

[12] We assume an MVC or similar architecture is being used here, or anything that clearly separates the UI from the rest of the system.

Referenced books

- "The Cucumber Book" (Wynne and Hellesoy)
- "Application testing with Capybara" (Robbins)
- "Beautiful testing" (Robbins and Riley)
- "Experiences of Test Automation (Graham and Fewster)
- "How Google tests software" (Whittaker, Arbon et al.)
- "Selenium Testing Tools Cookbook" (Gundecha)

Referenced articles

- "Model Driven Software engineering" (Brambilla et al.)
- "Continuous Delivery" (Humble and Farley)
- "Domain Specific Languages" (Fowler)
- "Domain Specific Modeling" (Kelly et al)
- "Language Implementation Patterns" (Parr)

> about the authors

Rudolf de Schipper has extensive experience in project management, consulting, QA, and software development. He has experience in managing large multinational projects and likes working in a team. Rudolf has a strong analytical attitude, with interest in domains such as the public sector, finance and e-business.
He has used object-oriented techniques for design and development in an international context. Apart from the management aspects of IT-related projects, his interests span program management, quality management, and business consulting, as well as architecture and development. Keeping abreast with technical work, Rudolf has worked with the StratEx team, developing the StratEx application (*www.stratexapp.com*), its architecture, and the code generation tool that is used. In the process, he has learned and experienced many of the difficulties related to serious software design and development, including the challenges of testing.
LinkedIn: *be.linkedin.com/pub/rudolf-de-schipper/3/6a9/6a9*

Abdelkrim Boujraf owns companies developing software like StratEx (program and project management) and SIMOGGA (lean operations management). He has more than 15 years of experience in different IT positions, from software engineering to program management in management consulting and software manufacturing firms. His fields of expertise are operational excellence and collaborative consumption. Abdelkrim holds an MBA as well as a master's in IT & Human Sciences. He is a scientific advisor for the Université Libre de Bruxelles where he is expected to provide occasional assistance in teaching or scientific research for spin-offs.
LinkedIn: *be.linkedin.com/in/abdelkrimboujraf*