9

te
testing
experience

The Magazine for Professional Testers

Advanced Testing Techniques

ignite
GERMANY 2010 Conferences Special

Test design for stubborn applications

Event handling in automated acceptance tests

by Alexandra Imrie & Markus Tiede

At the beginning of any test automation project for acceptance tests, the focus is usually on creating a set of running tests to cover the most critical or newest aspects of the software being developed. Such acceptance tests see the application as a black-box, testing the business logic through the GUI. Once a good base of runnable tests has been produced, however, it quickly becomes critically important to ensure that events that affect one part of the test do not lead to all other tests subsequently failing (due to "inherited" problems from the first failure), or not running at all (because the whole test has been aborted) . The discovery of errors in an application is a natural and desired effect of automated acceptance tests; however, the other aim of acceptance testing should always be to have the most comprehensive knowledge of the software quality. If 90% of the tests cannot run because an event occurred in the first 10%, this aim cannot be achieved. The quality will suffer as a result, and the costs to fix any subsequent untested errors will increase as more time passes between introducing the error and finding and resolving it.

A well thought-out system of event handling is therefore necessary to ensure that the quality of the whole software project can be monitored despite problems in individual areas. In addition, adding event handling to the project can make it easier to find and reproduce errors to allow them to be fixed in a short time. This article offers strategies and suggestions on introducing robust and understandable event handling in automated acceptance tests.

Definitions

The terms *error* and *exception* have specific meanings for software developers: *exceptions* can be handled and errors cannot be handled. From the perspective of the tester, this meaning holds true. Therefore, for the purposes of this article, we use the following definitions:

Event: Either an error or an exception; something that causes the test flow to deviate from the expected execution.

Exception: An exception results from a failed assertion in the test execution (for example, an expected value or status does not correspond to the actual value or status). With excep-

tions, the test can continue after some kind of "intervention" by an event handler.

Error: An error is an event that can't be dealt with in the test execution. Any following steps that are dependent on the success of this step cannot be executed.

It should be noted that what is an *error* for a developer (something that the software cannot handle) may manifest itself as an *exception* for the tester (something that the test execution can deal with), and vice-versa.

What types of event can affect an automated test?

Events always reflect a discrepancy between an expected state or behavior and the actual state or behavior in the application. The test defines what is expected, and these expectations are compared to the actual implementation in the application. An event can therefore be the result of one of three causes (see figure 1):

1. A mistake in the test specification (what is expected), for example a logical mis-take in the test or a misunderstanding between the test and development teams.

2. A problem in the test environment, including memory or performance issues, wrong paths for the environment, problems arising from other programs on the machine, database problems and runtime environment issues.

3. An actual error in the software; these are the errors that tests are supposed to find.

Figure 1: Error sources

© BREDEX GmbH
created with yEd
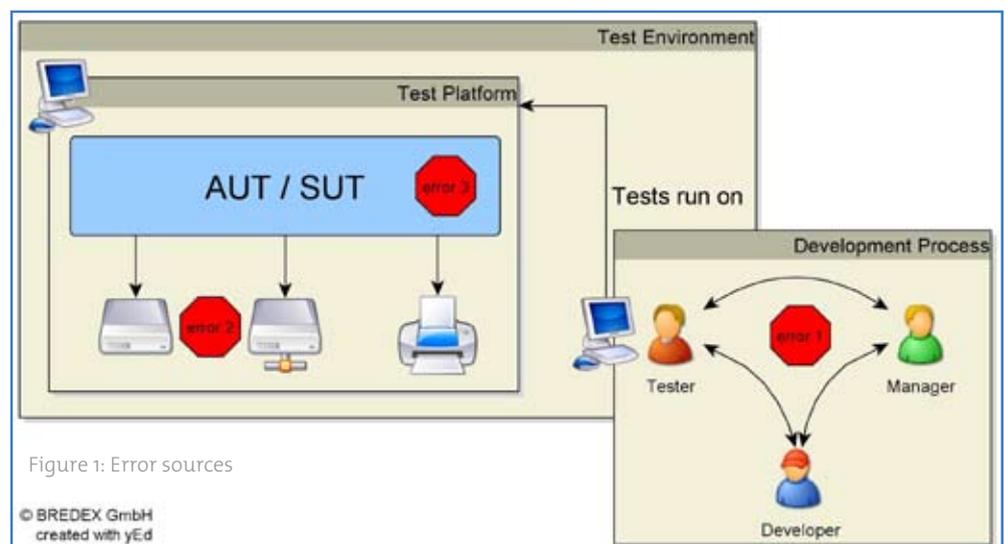
Minimizing errors that occur because of these first two factors is possible by adapting the test process and by setting up a stable environment (more on this later). Neverthe-less, regardless of the cause of the event, it is important to be able to react accordingly, depending on the type, location and severity of the event.

Events can manifest themselves in various ways in an automated acceptance test:

1. The expected state does not match the actual state:

    a. The data used in the test are not valid in the context of the application (for example, a menu path that the test tries to select doesn't exist).

    b. An assertion made by the test fails during execution (for example, a result calculated by the application).

2. The expected behavior and the actual behavior are different:

    a. The workflows defined in the test cannot be executed on the application (e.g. the test assumes that the workflow takes place in one dialog, but the workflow is divided over multiple dialogs in the application).

    b. The component in the software which should be tested cannot be found at runtime (this can either be a result of a different expected workflow or an unexpected event in the software).

Knowing what types of event to expect during test automation can help to plan and adapt the event handling system, so that maximal test coverage is attained despite any events which occur. This being said, preparing for unexpected events can be a challenge.

Planning for unexpected events

In any test, there are theoretically a number of things that could go wrong. Buttons that need to be clicked could be disabled; whole screens could be blocked by an error message and so on. Depending on the use case being tested, there could be some time distance (usually 1-3 actions) between an event occurring and this event adversely affecting the test. This makes searching for the cause of the event (and handling it, if possible) more difficult. Ideally, the test should be structured so that events are discovered as quickly as possible. In the example of the button, an enablement check (expected value: true) would suffice to prevent the event before a click has even been attempted. In other words, checks can be used to identify an upcoming event (a falsely disabled component) instead of producing an event later on in the test (the click was sent, but the dialog didn't close and the next action fails due to the dialog still being visible).

In the strictest case, this would mean performing checks before every single action. This does, however, require careful planning in terms of test design to maximize reusability and minimize effort. The advantage of this approach is that the events produced are more likely to be exceptions, which can be dynamically handled in the test (the test can react to a failed check and create the state or condition necessary to continue, much as a manual tester would do). Alongside this aspect of test design, a good structure of the test as a whole is critical to be able to cope with events.

Test design to support event handling: independent use cases

Designing tests so that individual use cases are independent of each other is a critical prerequisite for effective event handling. A test can only meaningfully continue after an event in one use case if the following use cases do not assume that this first use case was successful.

This means dividing tests into separately executable use cases, which have their own structure:

* SetUp: This ensures that the application is in the expected state for the use case to begin. The starting point for any use case is often a freshly started ap-plication. After making sure

that the application is ready to be tested, the SetUp can then continue to create the conditions required by the application to run the use case.

* Workflow: The whole workflow for the use case to be tested should follow the SetUp. The use case should have no prerequisites from any other use cases and should create no dependencies for further use cases.

* TearDown: Once the use case is completed, the TearDown brings the applica-tion back to the state expected as a starting point for the SetUp. This can in-volve removing any data created during the use case and recreating the default inter-face for the application.

Within the whole test, this structure results in a "handshake" between each use case. If one use case was successful, the application is already in the correct to state to begin with the second use case.

One question arising from this structure is how to manage data or preconditions required to execute a use case. In a use case to delete a project, the SetUp should ideally prepare the project to be deleted – although preferably not via the test automa-tion itself (i.e. do not write tests to create necessary data for other tests). Specifying tests which set up data is time-consuming (both in terms of test creation and test execution) and creates dependen-cies in the use case (if there is an error creating the project in the application, the delete action cannot be tested). In such cases, a concept similar to mocks and stubs is necessary. For applications which use a database, for example, a set of default data could be present in the test database. This data can then be recreated in the database using a simple script when necessary.

This being said, the paradigm create – edit – delete is common in tests and, depending on the project, it may not be desirable to separate all three use cases. In such situa-tions, it is important to be aware of the dependencies in a use case and to minimize their effects where possible.

Test design: handling events at the right level

Having designed automated tests as independently executable use cases, event handling can be added. There are two levels where it is reasonable to deal with events – locally and globally (see figure 2):

Local event handling can be used for exceptions which are ex-pected (known prob-lems or inconsistencies) and which can be resolved or ignored for the use case. The aim is to document the error, to dynamically "fix" it for the test and then to continue. For example: a check-box should be activated for the test to succeed. If the check-box is not activated, the error handling can activate it so that the test can continue.

Global event handling is used for the "unexpected" errors which are neither antic-ipated nor "fixable" in the test. Such errors usu-ally mean that the application is in an inconsistent or unknown state and therefore that the rest of the use case could be affected by inherited errors. The aim of global error handling is to docu-ment the error (screenshot, description of the error type and the use case it occurred in, log entry) and then to prepare the appli-cation for the next use case. By using the same TearDown func-tion as at the end of a successful use case, the same handshake between use cases is assured. The success of such global handling rests on the independence of the use cases – if the following use case assumes the successful completion of this failed use case, then more events will follow.

Figure 2: Local and Global Event Handling

© BREDEX GmbH
created with yEd
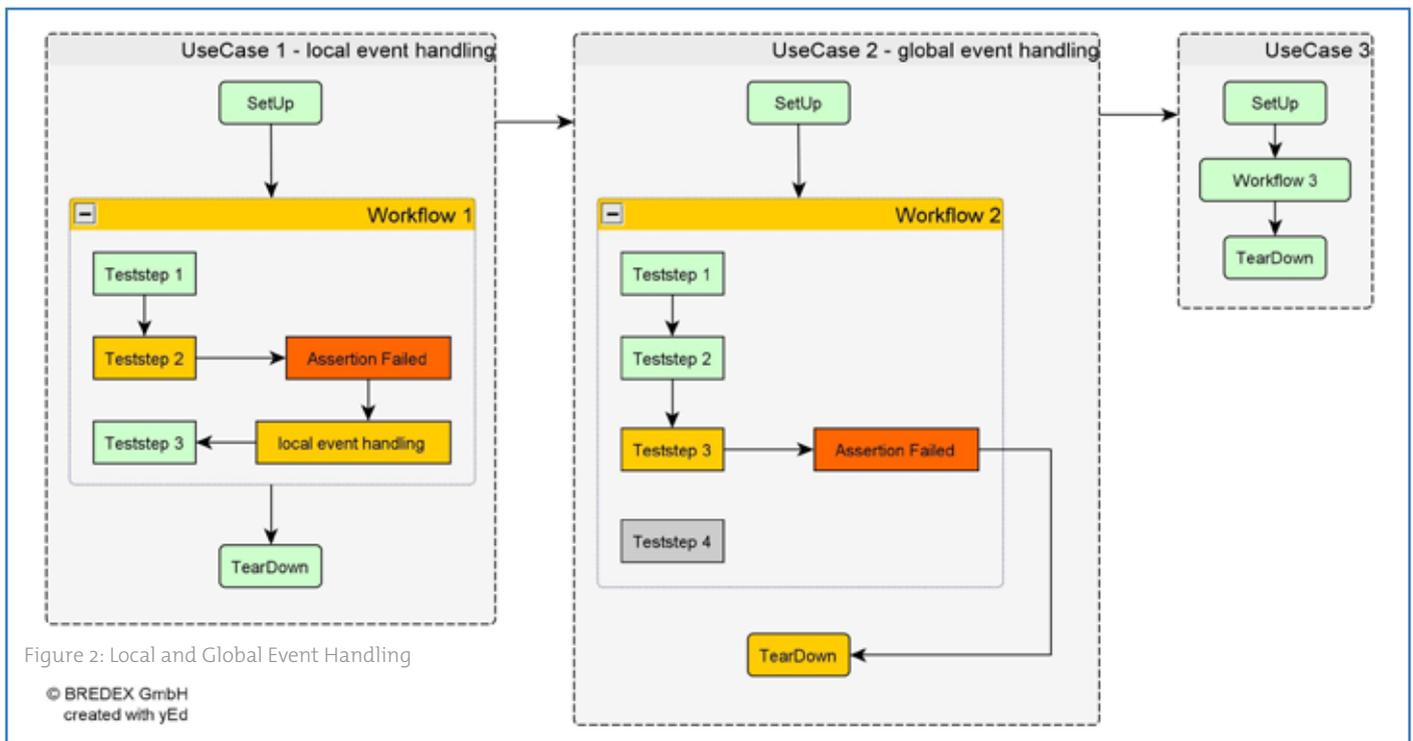
It can be tempting to try to react to every potential event locally; however, a good rule of thumb is to start with global handling (which ensures that all events are caught and that the test can continue) and to add local handling for manageable exceptions as it becomes necessary.

Can events be avoided altogether?

As mentioned above, events can come from various factors. Events resulting from actual errors in the software are unavoidable (if they were avoidable, there would be no need to test!), but events that come from the other two areas (from the tester and from the environment) can often be reduced. Events resulting from mistakes in the test specification can often be avoided by improving the communication in the test and development process. By inviting the test team to development meetings and by formulating requirements directly in terms of testable criteria, many misunderstandings can be discussed and the negative effect on the tests avoided. Encouraging the test and development teams to communicate changes and questions outside of formal meetings can also lead to fewer events from this area.

Events that occur because of the environment can be minimized by having a robust strategy for preparing necessary data and conditions for the test without using the test itself to set them up. A robust synchronization within the tests (flexibility if certain actions or processes take longer than usual) can also reduce the amount of unnecessary disruptions to the test. As well as synchronization, the tests also need to include as much "intelligence" as a manual tester would use to react to small changes or inconsistencies, plus an explicit understanding of how the application's interface and logic works (how focus works in the application, how the completion of processes is displayed in the interface etc.). In this way, tests do not rest on assumptions that may change or that turn out to be false, but are based on well-defined workflows and actions.

Dealing with errors in a project

Handling events in the tests is an important step, but once an error in the software has been discovered, there must be processes in place to handle these errors within the team. There are two main options for dealing with errors found by the automated tests:

1. Fix the error in the software instantly. This has the advantage that the fix is close to the development, which reduces the

cost to resolve the error and also im-proves the chance that the developer who introduced the error is still working on the code. It also means that the affected use case is up and running in the shortest time possible so that it can continue to test the software as intended. Because the time between finding and fixing the error is so short, there is no need to consider removing the test from the set of productive tests. The disadvantage of this approach is that the team can end up doing "error driven development", whereby normal tasks are left to one side to ensure that the automated tests are kept running. This approach is also unsuitable for errors that are not quick and easy to fix, but which are just the "tip of the iceberg".

2. Write a ticket for the error. This has the advantage that the resolution for the error can be planned better (which is useful if the fix has wider-reaching conse-quences or risks in the software). If a ticket has been written, it may be worth considering removing the affected use case from the productive tests and instead running it with "known error" tests (which is easy to do if the use cases are independent of each other). The disadvantage of this approach is an accumulation of tickets over time, while parts of the affected use cases cannot run until the error is resolved.

Conclusion

Event handling isn't necessarily the first thought when it comes to writing automated tests. However, as the amount of tests grows, and the reliance of the team on the quality monitoring increases, a well-thought out event handling mechanism can make the difference between comprehensive test results despite errors and a complete test fall-out after every single misunderstanding or event.

The implication of this is that the test tool used to automate the acceptance tests must support the mechanisms required to implement successful event handling. This includes being able to react dynamically to events in a variety of ways depending on the event type, level and the situation where the event occurs. It also means that tests should be under continuous scrutiny so that improvements to the test structure and additional local event handling can be added as necessary.

Biography

Alexandra earned a degree and an MA in linguistics from York University before starting work at BREDEX GmbH. She is actively involved in various roles at the company – alongside dealing with customer communication, support, training and demonstrations, she also enjoys representing the customer perspective in the software development process. She has experience of working and testing in both traditional and agile projects.

Markus is a software developer and tester at BREDEX GmbH. His main areas of expertise are the design and maintenance of automated tests and Eclipse RCP development. He is one of the core team developers for the automated test tool GUIdancer and holds a Diplom (German degree) in Computer Science from the University of Applied Sciences in Braunschweig.

ISTQB® Certified Tester-
Foundation Level
in Paris, in French

Test&Planet: votre avenir au présent.

Venez suivre le Cours « Testeur Certifié ISTQB – Niveau fondamental » à Paris, en français !

Test&Planet, centre de formation agréé DIF, vous propose ce cours D&H en exclusivité.

Apprenez à mieux choisir Quoi, Quand et Comment tester vos applications. Découvrez les meilleurs techniques et pratiques du test logiciel, pour aboutir à une meilleure qualité logicielle à moindre coût.

Possibilités de cours sur mesure.

Pour vous inscrire ou pour plus d'information: www.testplanet.fr

Test & Planet

Díaz Hilterscheid