

## Automated GUI Testing with GUIDancer

### Abstract

Automated GUI testing, although hailed as the solution to testing problems, has proven to not be as easy as was previously thought. This paper looks at the problems with automated GUI testing, and presents a different way of creating tests which results in the reduction of cost and effort in the test process.

### Introduction

As technology has developed, there has been a desire to move away from manual testing. Although thorough, there is no cost-effective way of regularly and reliably repeating ever-growing tests manually. In response to this, a huge market for automated test tools has sprung up, a market in which GUI test-tools are not underrepresented.

The market for automated GUI test-tools can be split into different camps:

- tools with which tests are essentially programmed
- tools with which tests are recorded in a running application
- tools which involve a mixture of the two
- other tools (e.g. GUIDancer)

Each method has certain advantages and disadvantages, most of which have been adequately discussed before. A quick recapitulation of the main arguments follows:

### Programmed tests

Although programmed tests are powerful, any test process which involves programming inevitably adds cost and time. The resources needed to write and test the code for the tests, as well as maintain and expand the tests are considerable, and programming may not necessarily (one could even argue, should not) be a part of the skillset of the testing team. Moving test creation or automation away from the testers wastes time and blocks resources. Even test processes with minimal amounts of programming, such as the development of keywords, force the testing team to wait for updates and maintenance, and add to the effort of the testing process. The splitting of the roles between developers and test-

ers also breeds inefficiency – test documents and test code both have to be created and maintained.

### Recorded tests

At first glance, recording a series of interactions in a running application and replaying it seems like a good idea, especially based on how easy it is to record the interactions. Using capture-replay tools also moves the centre of control more towards the testers. Where this approach falls short is in the loss of critical testing time before each version of an application is released. Test creation can only ever begin once an application is available, and even then, simply recording is not enough. Checkpoints have to be set, and interactions have to be parametrized. In most projects, test creation at this point is too late, and the tests quickly fall behind the application. Serious issues may not be found by the testing department, and the product quality falls or the release is delayed. Maintenance of recorded tests is also infamously difficult, and the scripts generated are of questionable quality. Test design should mirror the best practices of software design, and should be based on generic units and reuse. Clicking tests together results in very little reusability, and means that tests for new functions must essentially be built from scratch, instead of building on previous tests and reusing already existing units.

### The problem

The problem with this picture of automated GUI testing is that tests cannot be created early enough *by the test experts* (whose skillset does not necessarily involve coding/scripting). This leads to the test process being at a disadvantage because test creation starts so late (and can therefore not keep up with development), or involves too many resources. The costs are high and there is no chance of agile GUI testing.

### Requirements for a GUI test tool

In an ideal world, testers should be able to start with test creation at the same time as the development begins, using the requirements for the test design. Only in this way can each new version be tested thoroughly and efficiently. Test creation should be a cumulative process, with parts of existing tests being recycled to make new tests. This presupposes a certain degree of flexibility in the test design. GUI tests in particular should be flexible because of the frequency of changes in a developing application's interface.

### The solution

GUIDancer supports keyword-driven testing – tests are created from reusable modules (Test Cases). The difference to other keyword-driven tools is that the Test Cases are created without any programming or coding. The test specification is therefore the automation – Test Cases are executable without the need to implement them in code or script. Test Cases can be specified so as to be generic, and are platform and application-independent.

### Test Steps

When a manual tester carries out a step in a GUI test, he needs three pieces of information:

- Which GUI-component has to be tested?
- Which action has to be executed on it?
- Which parameters does the action have?

So, a typical test step to enter a username into a login field might look like this:

- Component: login field (text field)
- Action: enter text
- Parameter: <myUsername>

An automated GUI test only really needs the same three pieces of information, plus a way of assigning the component in the description to a component in the application (which the manual tester does with his eyes). A test step (the smallest unit in GUIDancer) is made up of these details: the type of component to be tested, the action, the parameter, and a name for the component which can be used to assign the specification to the application at a later point.

At this stage, two important things should be noted:

1. For the moment, only the action needs to be fixed. The component(s) the action will be executed on and the data for the action can be defined and changed later (in essence, parametrized). This keeps the test as general as possible, making it easier to reuse, adapt and maintain.

2. The application under test does not have to be available to begin with testing. Test specification is carried out in *GUIDancer* using interactive dialogs from which all supported components and actions are offered. Because the test can be created in an abstract way, it is easy to add details later.

### Test Cases

Test Steps form Test Cases, which are the reusable modules (or keywords) in *GUIDancer*. They can consist of as many Test Steps and referenced Test Cases as necessary, and can be reused in as many other Test Cases as desired. In this way, the hierarchy and design of the test are all in the hands of the tester. Reusing Test Cases has the advantage that test creation is exponential. New tests are easy to create from an ever-growing library of modular Test Cases. Maintenance of tests is quick and easy because Test Cases are referenced, not copied. A few central changes can update a whole test. So changes in the AUT or to the requirements do not lead to re-specification of tests.

### Execution

Once the application under test becomes available, all that remains is to join the test specification to the application. This is done in the *GUIDancer* object mapping mode. In this mode, information on components is collected from the application using a key combination. Components are viewed as objects, and so various details are stored, including the name (if there is no name given, *GUIDancer* creates one), the place in the hierarchy and the context the component is in. This information is used to create an intelligent component recognition system during test execution. The component which has been “collected” from the application is then assigned to the user-defined component names from the test specification. This lets *GUIDancer* know which components to test at each point in the test, thus performing the same intelligent mapping as a tester’s eyes – a relationship between the test description and the actual AUT.

### Benefits for the test process

Specifying tests with *GUIDancer* plugs the time gap between development and testing. Continuous test development allows continuous integration – meaning that issues are found earlier, when they are easier and cheaper to fix. Not only that, but the tests themselves are less of a strain on the budget. Written and maintained by the test team, there is

no need for “automation” of test designs. The specification *is* the implementation. A tick in the box for careful planning and use of resources. The modular test design provides all the power and structure of a good piece of software code – reusability and maintainability – but without the expense of coding. After all, how can you advocate good practices in your software development, but ignore them for your tests? GUIDancer promotes and supports good test design, and brings flexibility and strength to tests.

### Conclusion

Waiting for an AUT to become available or wasting valuable time and resources are not the answer to automated GUI testing. Moving away from recorded or programmed tests makes the test process more stable, more efficient and, ultimately, reduces the cost while increasing the quality of the delivered product.

GUIDancer makes agile testing possible for GUI's. Figures from projects show that using GUIDancer can keep the cost of automated GUI testing under the 10% mark in terms of total project cost, while delivering a high quality product to the customer.