

Automating GUI-Level Testing in XP

Markus Mannevaara (markus.mannevaara@hut.fi)

Abstract—In this paper, automation of GUI-level testing is considered in the context of Extreme Programming (XP). The challenges which make GUI-level testing difficult are investigated. Various approaches to automating tests are presented, and evaluated for suitability in XP programmer- and customer tests. For programmer tests, extending currently available unit testing frameworks with software interfaces for manipulating GUIs, is found favorable over moving to traditional GUI test tools. For customer tests, the problem of having a readable yet executable format is discussed. Fit is presented as a possible solution.

Index Terms—GUI Testing, Automation, Extreme Programming, Acceptance Tests, Functional Tests, Customer Tests, Unit Tests, Programmer Tests, GUI Test Automation

1. INTRODUCTION

In this paper I present various approaches to automating GUI-level testing¹, and discuss how well they are suited for use in Extreme Programming (XP). I try to clarify my discussion by first analysing GUI-level test automation as a separate problem domain, and which explicit or implicit criteria Extreme Programming places on test automation and testing. Using these analyses as a basis, I hope to make the argumentation more clear for both myself and the reader. The purpose of the paper is to allow focusing research and application development in the field of GUI-level test automation.

1.1. Background

Extreme Programming (XP) defines two forms of tests, which are named according to their owners: *customer tests* and *programmer tests*. Customer tests specify acceptance criteria for stories that are to be implemented, and programmer tests are intended as a design tool and safety net for the programmer. XP places emphasis on that these tests be automated, and written before the code they exercise. (Beck 2000)²

Programmer tests are typically small tests that each exercise a minimal set of functionality. A large collection of these tests is built up incrementally during an XP project. Unit testing with xUnit-style testing frameworks (see e.g. Jeffries (2004)) has proven very popular, successfully allowing programmers to use tests as a design tool by writing them up front. (see e.g. Beck (2002))

Customer Tests typically apply to larger sets of functionality than programmer tests, and, although the xUnit frameworks are not intrinsically limited to programmer tests, they can only exercise software interfaces. GUIs do not necessarily expose a software interface suitable for use in tests, making GUI test automation nontrivial.

¹I use the term *GUI-level testing* to refer to all testing that is performed through a graphical user interface.

²Previously, customer tests were known as acceptance tests, and before that as functional tests. Programmer tests were previously known as unit tests. The used terminology is from (Jeffries 2001).

To address this problem, several vendors have developed GUI test automation tools, with facilities to automate tests run against a GUI. Even though a large number of these tools exist, GUI testing is not fully familiar. This study investigates GUI-level testing in XP, considering various approaches and their suitability to programmer- and customer tests. These approaches are gathered from literature presented in the XP/Agile community, and material from the “traditional” testing community. In particular, application of the mentioned GUI test tools is considered.

1.2. Research Problem

This study is organized around these four questions.

- 1) What are the problems faced when automating tests against GUIs?
 - Are they primarily practical problems, or is there something fundamentally difficult with GUI-level testing?
- 2) What approaches exist to address these problems?
- 3) How would these approaches work in XP?
- 4) What could be done to improve the situation?

1.3. Objectives

The objective of this study is to further allow focusing research and technological development. Hopefully this can be achieved by investigating the problem domain to find what stands in the way of successful automation, and what directions can be taken to improve the situation. As a part of the analysis, I will look at general issues with GUI-level test automation and problems with it from other software development practices. I will also look at XP to find out where GUI-level testing surfaces.

1.4. Scope and Research Methods

This study is about automation of GUI-level testing. In particular, I will concentrate on the automatic execution of scripted tests, leaving out for example model based testing and automated test case generation.

As this is a pure literature study, any conclusions I reach will be based on my judgement of methods presented in other research, and I will try to present enough evidence for the reader to be able to assess my conclusions. Hopefully the research will allow future research to concentrate on approaches which show promise.

1.5. The Structure of this Paper

Following this section, the study begins with an analysis of GUI testing, automation of it, and problems therein.

Thereafter, in section 3, follows a discussion of the special properties of testing in XP, and how they affect the choice of capture replay versus manually written tests. Section 4 looks at programmer tests in particular, and section 5 looks at customer tests. The paper concludes with a summary of findings.

2. GUI-LEVEL TESTING

Before looking at the approaches to GUI test automation, I will consider GUI-level testing more closely. Is there something fundamentally difficult with testing GUIs and, in particular, automating these tests? How do GUIs as testing targets compare against other targets, such as software interfaces? To start out, we need a definition of GUI-level testing, and GUI-level test automation.

2.1. What is GUI-Level Testing?

A very down-to-earth definition is suitable for our purposes. The GUI

- receives input from the user, who uses input devices such as the keyboard or mouse.
- presents information for the user in a graphical form not restricted to a grid of textual characters.

This definition of a GUI includes standard user interfaces, written for any operating system. It also includes less traditional user interfaces, such as Web pages or multimedia presentations. Although they are technically very different, they all present the challenges to test automation, which are discussed in this paper.

According to Bach (1996), a test can be considered as "...a sequence of interactions interspersed with evaluations." By GUI-level testing, we refer to testing which

- exercises the target through simulating the input performed by the user (interactions), and
- makes assertions (evaluations) about the target based on information extracted through the GUI.

Sometimes, assertions are made against the back-end of the system (gray-box testing), but nevertheless, we need the ability to make assertions through the GUI.

2.2. What is GUI-Level Test Automation?

For the user, the representation of the GUI consists of the pixels on screen, and the input devices she uses to enter input. For the purposes of automated tests, however, it is impractical to operate on this level. Usually, the GUI has a logical representation, from which the graphical representation is drawn. Input devices trigger events against this logical representation. A couple of examples: A widget presenting some text on screen has a logical representation from which the text can be extracted directly. A button on-screen is represented by a logical button, which receives a click-event when the user clicks the button with the mouse.

For the purposes of testing, it is more practical to operate against the logical representation of the GUI. There may be cases when one wishes to test that the exact graphical presentation is correct, for example when writing custom widgets. Usually, however, we rely on the GUI-engine to draw

elements correctly for us, and perform our GUI-level tests against this logical representation.

This definition of GUI-level testing does not depend on the type of testing performed. In terms of XP, this means the GUI-level tests may be customer tests or programmer tests. At any rate, an automated GUI-level test is a test that exercises the application in the above described fashion, and produces a boolean statement as to whether the test passed or not.

2.3. Exercising the GUI Through Automation

Given that we have access to the logical representation of the GUI, our test-automation problem has not yet been solved. The primary purpose of the logical representation is to carry information optimized for the GUI-engine to be able to draw the GUI onto the screen, and receive input. The interface is not designed for test automation, whose requirements are slightly different.

As mentioned earlier, this is why GUI test tools have been developed. Let us look more closely at what services the GUI test tools provide.

2.4. Identifying Elements

For the purposes of testing, we need a mechanism for precisely finding the element we wish to manipulate. In essence, this is a search operation performed against the logical representation of the GUI.

Of course, the GUI-engine has an operation like this. When the user clicks somewhere using the mouse, the corresponding element must be found. This is, however, performed based on pixel coordinates. When writing an automated test, we prefer to find a button based on logical information about it.

Several GUI test tools also provide the possibility of specifying the search criteria for a widget even though the widget may not be on screen at the moment. When performing several operations on the same widget, which may disappear and reappear during the test, it is useful to only have to find the widget once.

Once we have pinpointed the element, we are faced with the smaller problem of parsing the information that is usually presented in natural language. Often, however, regular expressions (or similar) can be used to extract the information.

2.5. Synchronizing with the GUI

When performing operations on a GUI, the effects of those operations are usually not instantaneous. There may be a significant delay before the GUI is updated to reflect the changes which are caused by the operation. Even though it may be possible to gain synchronized access to the GUI components, the business logic of the application may be running in its own threads. In general, the GUI may change at any time during the execution of a test.

We need some mechanism that allows synchronizing with the GUI. After performing an operation we need to be able to wait for the operation to complete, before making assertions about the state of the program. This is different from testing software interfaces, where operations are usually synchronous;

once a method-call returns, the testing target is ready for further interactions. With a GUI, once a button has been clicked, the GUI may be in a changing state for a significant amount of time after the test is ready to proceed.

2.6. Services Provided by GUI Test Tools

The challenges presented in this section are addressed by several GUI test automation tools. To repeat, these services include

- attaching to the logical representation of the GUI,
- finding elements in the logical representation, and
- synchronizing with the GUI.

In addition to these services, test automation tools provide several other services, not directly related to exercising the target. Some of the most important one are

- allowing creation and editing of tests,
- executing tests,
- tracking the pass/fail status of tests, and
- producing execution reports.

I have chosen to refer to the latter set of services as *testing framework services*, and the former ones as the *test target API*. The exact shape of these services may vary between tools, but they exist in one form or another in each tool.

To see why the test target API is a relevant part of a GUI test framework, contrast GUI testing to testing software interfaces. Software interfaces were designed to be used from software, allowing tests to be formulated directly against the interface itself. No test target API -layer is needed in between.

Since the tests are formulated using the test target API, it is likely that it will greatly affect the way in which GUI-level tests are written. Especially in the case of manually written tests, the design of the API is likely to affect the expressiveness and clarity of the test, as well as the productivity of the test writer.

The xUnit-frameworks, which have proven so popular for implementing programmer tests in XP (see section 4), are in fact simply providing an implementation of the *testing framework services* mentioned above. The test target API is not needed, since the programmer tests are traditionally written directly against the classes to be tested. It could be that this is one of the success factors behind xUnit.

3. GUI-LEVEL TESTING IN XP

In this section, we will look at a two general approaches to creating GUI-level tests: recording and manually writing them. First, however, we will look at testing in XP in general.

3.1. Testing in XP

A fundamental principle in XP is “Work with human nature, not against it.” (Beck 2000) Instead of trying to enforce good quality through rigorous discipline, we try to arrange the project and daily work so that creating quality becomes less of an obstacle. The thought is that approaching quality like this, using conventions that can be easily followed on an individual level, the barrier to apply these conventions will be lower. This, in turn, is expected to produce a better net effect than

an approach which is rigorously disciplined but requires much more effort.

In XP developers are not as worried about coverage as in traditional testing. This is due to the incremental approach of XP; customers and programmers have the freedom of adding to their respective test suites whenever they see something that should be there. The principle is embedded in Beck’s words from XP Explained: “Every time [the customers] think of something concrete the program should do, they turn it into another piece of confidence that goes into the program.” (Beck 2000) It is forgivable and likely that something is overlooked during the initial formulation of tests or requirements for some feature. Without the option of adding or changing tests later on, time must be (often ineffectively) spent to scrutinize documents to see if anything has gone missing, before starting implementation.

XP strives to automate all tests. In the face of constant change this may mean constantly changing tests. The tests must be designed with this in mind; whatever format the tests are created in, must allow easily changing the tests.

In addition to ensuring that the program does what it is supposed to, the purpose of tests is also to increase confidence in the code. From the perspective of programmers, Beck writes: “So every time [programmers] think some code is going to work, they take that confidence out of the ether and turn it into an artifact that goes into the program.” Beck (2000) Confidence is increased for whoever is able to read the result of a test, and believe in it. In order to believe in the result of a test, one must understand the test.

To understand a test means being able to read it and agree that it verifies something of value. It is not enough that a test can be executed; it must also serve a documentative purpose. This will be one of the criteria in investigating approaches later on.

In the remainder of this section, we will consider two general approaches to creating tests, and how well they are suited for use in XP.

3.2. Capture Replay

Capture replay³ is a method for creating automated test scripts. The program records interaction with the GUI, and is able to play them back later, reporting a failure if some action cannot be played back. In addition, tools may support adding checks to these scripts to perform validations on the GUI.

Even though there are several tools for this purpose, software testing experts claim this is not a successful method for creating automated tests. For example, in the book “Software Test Automation” (Fewster & Graham 1999), Fewster and Graham have an entire chapter titled “Capture replay is not test automation”, where they give examples of situations how capture replay tools fail to deliver. Also for example (Finsterwalder 2001) and (Kaner 1997) comment on problems with

³Sometimes this is referred to ‘record and playback’ or ‘capture and playback’. I have chosen to talk about ‘capture replay’, from Fewster & Graham (1999).

capture-replay tools. Kaner identifies Capture replay as one of the most common reasons for test automation to fail.

Considering capture replay in terms of XP, I can identify these reasons why they are unsuited:

- require much work to change,
- require maintenance more often,
- are hard to understand, and
- are forced to lag behind implementation.

In the following, each reason is considered a little closer.

Captured tests require much work to change. In the context of a single test, problems arise when one step needs to be changed: the step would need to be re-recorded. If the same step exists in many steps, that step needs to be re-recorded in each test.

Captured tests require maintenance more often. Criteria by which the tool identifies elements are controlled by the tool itself, and as such are likely to be more sensitive to changes in the application than manually written tests.

Captured tests are hard to understand Captured tests do not include any information about intentions of the test, or semantics of the test, since the test is just a mechanical recording. This is very undesirable in XP, since the tests ought to function as some kind of documentation as to what the target application is intended to do.

In particular, it is not possible for the customer to create captured tests at will in order to describe new intended functionality of the system.

Captured tests lag behind implementation. Because they are recorded, captured tests cannot be completed before the target of the test is completely finished. This forces captured tests to lag at least one step behind the actual implementation, making it impossible to practice test-driven development with the captured tests, making them unsuited for programmer tests.

It might be tempting to solve the problem of hard-to-understand tests by including a description of the test in natural language together with the test. Either in the comments of the test script, or in a separate context somewhere. This means effectively duplicating the test script, and if we have an alternative of maintaining only one representation of the test, we choose that one.

In general, captured tests tend to be brittle, and require much maintenance work. They do not lend themselves to redesign through refactoring, and in the face of constant change, their expected lifetime is short.

Having said all that, in his paper "Agile Regression Testing Using Record and Playback", Gerard Meszaros recognizes some special testing problems that can be solved using record and playback. However, the examples presented are projects in which no automated tests have been developed during the project, and as such are not relevant in XP projects. (Meszaros 2003) Kaner (1997) also recognized the value of capture replay as a means for understanding how the testing tool interprets certain interactions with the GUI.

3.3. Manually Programmed Tests

Should one choose to write test scripts manually, test creation becomes a programming task, subject to the same good

and bad practices as normal programs. The thought of this is unattractive, since intuition tells us that in order to achieve full test coverage, the test project has to be proportional in size to the target project. Developing an extra software application to be maintained in parallel with the actual one does not sound reasonable.

However, support for this thought exists in literature, and many experts, including Bach (Bach 1996) and Fewster and Graham (Fewster & Graham 1999) advice in this direction. Cem Kaner (Kaner 1997) puts it tersely when saying: "Recognize that test automation development is software development."

In practice, a set of test scripts start to resemble a software application as soon as frequently used operations are abstracted into their own function libraries to be shared across tests. Kaner calls this framework-based testing, and lists it as one of his "Strategies for Success." (Kaner 1997) He also anecdotally recognizes that the number of lines in test code may be equally large or larger than that in the target. This provokes me to question the sanity of such work.

Considering the nature of test automation as software, and contrasting this to the target software being tested, there is reason to believe that the test code is easier to produce, even though it may be larger in size.

The images evoked by producing two proportionally sized pieces of software is easily interpreted as producing two solutions for the same problem. However, it is important to realize that the automated tests do not solve the same problem as the target software, even though it requires some knowledge of the target application's problem domain, and perhaps even some of the same calculations. The test software merely exercises the target application using examples of calculations for which the results are known. Even though this may translate into more code, these code-lines are not necessarily more time-demanding to produce.

Problems that need to be solved by the actual application but which are of no concern for the tests include for example resource management and business calculations. Instead, automation solves problems relating to attaching to the user interface, and correctly formulated script-sequences. Attaching to the UI is a problem, whose solution can be transferred from one application to another, thus amortizing the cost of developing it.

All of this is not arguing that manual test automation comes without a cost, but rather that even though the code produced is a form of software project, some significant problems of software projects are not present in test automation projects. The lines-of-code measure is not a useful comparison between the test code and the target code.

With the backing of experts and the above reasoning XP's goal of full test automation seems defensible. Especially considering that manually written tests have the potential of serving a descriptive purpose.

4. APPROACHES FOR PROGRAMMER TESTS

In this section, we consider various approaches to helping out the situation of programmer tests for GUIs in XP.

Programmer tests can be defined as: “A programmer test assists programmers in development, without specifying how.” The purpose of programmer tests, and the related Test-Driven Development (TDD) (Jeffries 2001), is to help design and implement functionality, and to verify correct operation after implementation.

Usually, programmer tests take the form of unit tests⁴, which exercise a “single unit of code.” For my purposes, a sufficient definition of a unit is merely that it is small, and that it can be tested independently from other units.

Notice that there are two concepts at hand now, programmer tests and unit tests. In this study, I use the term unit test exclusively to refer to xUnit-style unit tests, and programmer test when I do not wish to make a distinction about how the test is done.

Among the challenges for doing programmer tests for GUI components, we find:

- *Test-Driven Development.* Doing TDD, the test for a unit of code is written immediately preceding the implementation of that unit. (Beck 2002) In his article “Aim, Fire!” (Beck 2001), Kent Beck identified that he finds it hard to write tests up-front for GUIs.
- *Isolation.* Preferrably, we wish to test each unit of code individually, without interference from other units. This lets us avoid complex setup procedures, and keeps the test simple. For GUI components, we wish to be able to test a single GUI component without having to setup the entire business logic that goes behind it.

4.1. Thin GUI

If testing code in the GUI is hard, then keep as much code as possible in other parts of the application, so that very little functionality has to be tested through the GUI. While not helpful strictly for testing GUIs, this advice is helpful for solving our testing problem in general. The XP rule states: “Test everything that could possibly break.” In terms of this advice, we can formulate the optimal solution as “Make the GUI so simple it cannot possibly break.” (Adapted from Koss (2002))

Various implementation strategies exist for decoupling the GUI from the back-end logic, for example the Model View Controller- or Facade patterns.

This strategy strives to reduce the need to test complex business logic in GUI-level tests. The business logic can then be tested through software interfaces, which, for reasons mentioned before, lend themselves to easier access through automated tests.

Considering programmer tests, this advice is particularly helpful, since it keeps more functionality in the back-end logic, and allows more tests to be written without the involvement of a GUI. It is also possible to perform unit tests against the GUI, as discussed in the next section.

For customer tests, this advice is maybe not as helpful, since there is often a desire to test the application fully installed.

⁴The definition of *unit testing* in XP is somewhat different from the traditional definition of unit testing. In this paper, I use the definition by XP, which refers to xUnit-style tests.

However, keeping as little logic as possible in the GUI may allow implementing some customer tests directly against the software interface of the back-end. This is not only more convenient, but also more reliable, since the tests will not be sensitive to defects in the GUI. As noted by Finsterwalder (2001), the GUI tests then simply need to test a few cases to verify that the “GUI is attached correctly to the underlying functionality.”

This approach does not, however, make it easier to create those GUI-level tests which we still want to create.

4.2. Using GUI Test Tools

In principle, it should be possible for programmers to use traditional GUI test tools to write programmer tests. These tools have a built-in test target API, which can be used to manipulate GUIs. It is likely that a tool exists which is able to exercise the particular platform on which the application is being developed. There are, however, a number of problems related to this.

The primary problem is that these tools were designed for testers and QA teams: not XP developers. The basic notion is that tests using GUI test tools are larger and fewer than the unit tests that usually function as programmer tests.

GUI test tools often require that tests be written in some proprietary language, and the test scripts can usually be edited only through the tool itself. The proprietary language is likely to be less expressive than standard programming languages, and the editing facilities of the GUI are not likely to accommodate the programmers’ needs as well as their regular programming tools. These issues, although mundane, are likely to put off programmers at least to some extent.

A final argument against using GUI test tools is the white-box nature of programmer tests. Since the tool operates entirely from outside the program, they are less suited for unit test-style development. As a concrete example, the tools aren’t likely to be able to instantiate a single GUI component and test it in isolation. This will force programmers to consider larger wholes when writing tests, making Test-Driven Development more difficult. (As Kent Beck says, we would like to be able to take tiny steps. (Beck 2002).)

4.3. Unit Testing GUIs

Since test-driven development is so popular among Extreme Programmers, and since said programmers are so fond of the xUnit frameworks, it is worth investigating how unit testing a GUI using xUnit would work out. The challenge for unit testing GUI components are

- accessing the GUI from the test,
- enabling test-driven development, and
- testing the user interface in isolation.

In Extreme Programming Installed (Ron Jeffries 2000), originally from (Wake 2000), Bill Wake presents a Test-Driven Development episode, where the GUI for a search form is developed. Mock Objects (See e.g. Dave Thomas (2002)) are used to replace the actual back-end system. Although it is a prepared example, it demonstrates that test-driven development is possible for GUIs.

However, the example has received some criticism, since it keeps and publicly exposes references to individual components in the GUI, merely to let the tests interact with the GUI. The application itself does not require exposing these references, nor keeping references to them after their creation. Considering more complex GUIs, this may involve publicly exposing a significant number of references only for the purposes of testing.

In Swan (2002), Andrew Swan presents another similar method for GUI-level testing, where he has substituted the explicit references with named GUI components. In his tests, he uses methods that recursively search the GUI tree for components based on their name, and performs operations on them.

There is no reason to believe that Swan's more refined method would not scale up for more complex user interfaces. In his article, he also claims this mechanism functions as a base for acceptance tests, but does not elaborate further on it.

Considering the challenges, the examples successfully demonstrate test-driven development, and testing the UI in isolation. Accessing the GUI from the test doesn't require much work, since the tests and the UI are written in Java. However, this situation might be a little trickier with GUIs written in other languages. In general, it is a matter of having to implement a sufficient test target API, as discussed in section 2.6.

Ready-made test target APIs exist for testing GUIs through xUnit. HttpUnit (Gold 2004) and JWebUnit (ThoughtWorks 2004) allow testing web user interfaces, and for example JfcUnit (Caswell et al. 2004) allow testing Swing interfaces from xUnit.

4.4. Separate Test Framework and Test Target API

Considering the approach presented in the last section about unit testing GUIs. Swan (2002) has implemented methods such as `findMenu()`, which recursively search the GUI for the desired GUI components. What Swan has done is implement a part of the test target API discussed in section 2.6. Notable is the fact that the API has not been provided as a part of the testing tool.

As discussed in section 2.6, the xUnit-style tools are really stripped down versions of testing tools, providing only the testing framework services. The idea is that these could be extended with separate test target APIs, which could be used to drive automated GUI-level tests. This has already been done to some extent with JUnit. JUnit has a few extensions for testing web pages (HttpUnit, JWebUnit), and Java GUIs (AWTUnit, JFCUnit, and SwingUnit).(refs, refs, refs)

A number of immediate benefits present themselves with this approach:

- Knowledge is transferrable. Since all various test targets are being tested using the same basic framework, test writers can transfer part of their immediate test writing knowledge to new types of testing targets.
- Easy extendability. Since the tests are written in a standard language instead of a proprietary language, new test target APIs can be implemented to be able to run tests

against new kinds of targets. This also means that the extensions can be developed by third parties, for example the vendor of the GUI itself.

- Familiar language. As Cem Kaner says, “[Test code] is code, even if the programming language is funky.” Kaner (1997) Test scripting languages are often less expressive than “real” programming languages, making test programming less productive. Also, standard languages are probably familiar to developers as well, lowering the barrier to test.
- Familiar work environment. Since tests are being developed in the same language as the code itself, they can be developed using the same tools.

If testing experts recognise that GUI-level tests are best developed manually, like software, it seems natural that that test development be moved away from the realm of test tools, and into the realm of development tools. Therefore, I consider the development of these test target APIs for xUnit and the like, as one of the most interesting development areas within XP testing.

What about the testers who are not developers? I believe this is not a problem, because as recognized by Bach (1996), test automation is most successful when the automated tests are being developed by a separate team than who do the manual testing. After all, manual testing is what testers do best.

5. APPROACHES FOR CUSTOMER TESTS

Customer Tests are owned and specified by the customer. They exist to verify the correct implementation of user stories, and to give both the developers and the customers confidence in that the code satisfies the customer's needs. (Beck 2000)

Customer tests are usually specified at a higher level, each test embodying something concrete that the application should do. This often means that the customer needs to operate against more wholly installed applications.

Customer tests do not necessarily have to operate on the user interface level; this depends on what the customer desires. If GUI-level tests are required for the customer to feel sure about what she is receiving, then the tests should be implemented on the user interface level. In many situations it may be more practical to specify tests in terms of business logic, which may allow executing them directly against the software interfaces of the application.

Considering the audience of customer tests, the problem of specifying the test is different from that of programmer tests. Since the tests are read (and written) by customers as well as programmers, software code is not necessarily an option. If the customer is not able to read the tests, part of the tests' confidence-raising ability is lost. On the other hand, the desire to automate all tests creates the need to specify the test in an executable format.

One obvious solution is to maintain a natural language representation for the customers together with an executable representation for the development team and the test execution framework. This in turn, is in conflict with the “Once and

only once” -rule of XP.⁵ It is therefore desirable to have one common representation that can be read and understood by customers, written by someone on the project team, and executed by an automation engine.

In addition to the general challenges on testing in XP, we identify the challenge of *readability*; that we need a format which can be understood by the customer, the programmers, and the test tool.

5.1. Using GUI Test Tools

The arguments for tests as software applications have been developed by test automation experts (e.g. Kaner (1997) and Bach (1996)), who were using GUI test automation tools. Bach recommends that a separate automation-group be set up for creating the automated tests, since the task of automating tests is too challenging for testers to do part-time. Having customers face these problems on an XP project is probably too much to ask.

Recognizing the need for expertise in automating GUI-level tests, is it possible to rearrange the work so that customers still specify the tests, but relieving them of the technical burden of automation? We still wish to avoid having two parallel representations of the test, because these are bound to get out of sync.

One alternative is the arrangement suggested in Testing Extreme Programming (Crispin 2002). A dedicated tester in the XP-team is in active discussion with the customer, and takes care of implementing customer tests as executable tests. The customer pairs with the tester any time she wishes to have a test automated, and together they take care of scripting the test.

Another possibility is finding a format for tests which can be used by customers to express tests.

5.2. Finding a Format for Customer Tests

Given that program code is not a viable format for customer tests in a general situation, and we still need a format that is also executable, what would be a good format? Some testing tools use visual formats such as collapsible trees, which I have not investigated here. Some tools (such as Canoo Webtest (AG 2004)) propose writing tests as Xml, but it is questionable whether the syntax of Xml is any simpler than that of programming languages.

A very simple format that is surprisingly powerful is the table. The idea is not a new one; the idea has been around in the form of data-driven testing for long. (See e.g. Kaner (1997) for a description.) Tables can be used to represent a surprisingly wide range of tests. The rows in the table do not necessarily have to be test steps. The rows may also represent for example combinations of inputs and outputs.

In the agile world, Ward Cunningham’s *Fit* (Cunningham 2004) is one incarnation of tests described in tables. The tables are stored as Html, with the intention to make them easily editable. Each table represents test data, and is related to a

fixture, written in a programming language such as Java, which takes care of executing the test described by the data.

In terms of customer tests (for which Fit is designed), Cunningham says: “I would now describe the act of [customer test] development as one of joint authorship where the customer and developer collaborate in the writing of satisfied tests.” It is a reasonable assumption that a customer organization has experience in document authoring, which may thus be a more welcome responsibility than test automation. Therefore I find that the Fit approach has potential.

With the customer responsible for authoring the Html tables, and the developers responsible for authoring the fixtures, this may be a very functional division of responsibility. It is unclear, however, whether the fixture development is as straightforward as implementing scripted tests. In Muckridge & Tempero (2004), an experience report about moving from specifying customer tests in Xml format to a Fit-based approach, the authors report: “It is straightforward to extend the fixtures that are supplied with Fit, as well as to make up new types of fixtures.”

There is one problem, however: Fit is not a GUI-testing framework, and does not provide test target APIs for testing GUIs. With independent test target APIs, as described in section 4.4, this is less of a problem, since the same test target API can be used from Fit and xUnit-frameworks.

Although it is the most different approach to organising the test work, it is the approach that shows most promise. If the division of work can be implemented successfully, this may be a format in which the customers can be brought into the activity of test development, and in which the customer tests have their confidence-raising ability.

6. DISCUSSION AND CONCLUSIONS

This paper has been a survey of approaches to performing GUI-level tests. Some of them have come from the agile community, some from other communities. The suitability for XP has been evaluated for each of the approaches.

Not surprisingly, the approaches designed with XP in mind, are more suited for use in XP teams than approaches designed with other audiences in mind.

Programmer testing of GUIs is possible, but would benefit from having proper test target APIs to efficiently be able to access the GUI. GUI test tools provide such APIs, but are unlikely to appeal to programmers. The tools are designed for testers, and do not provide as rich facilities for development as the tools programmers are used to.

The largest challenge for customer tests in XP is successfully involving the customer in test development. It seems unlikely, that the customer is willing to tackle the complex tasks related to test automation, but the customer does possess the knowledge of the business logic. Fit approaches this with a mechanism for dividing the responsibility so that the technical issues of executing tests are given to developers, while customers are left with the responsibility of describing tests. Fit suffers from the same lack of test target APIs as programmer tests, which is probably the largest obstacle to driving GUI-tests from Fit.

⁵The “Once and only once.” -rule is anecdotally referenced in Beck (2000), but is discussed in detail elsewhere, for example in the C2 wiki at <http://c2.com/cgi/wiki?OnceAndOnlyOnce>.

Traditional GUI test tools do not seem to have a natural place in XP. Tests being software, developers find it limiting to use other tools than development tools for the job. Again, tests being software, customers are likely to be put off by the complexity of GUI test automation.

Development of separately distributed test target APIs, which can be used from xUnit tools or Fit tools, has the potential of improving the situation significantly.

ThoughtWorks (2004), 'jwebunit homepage', <http://jwebunit.sourceforge.net/> (2004-05-02).
 Wake, W. (2000), 'The test/code cycle in xp: Part 2, gui', *xp123.com (also in Extreme Programming Installed)* .

REFERENCES

- AG, C. E. (2004), 'Canoo webtest homepage', <http://webtest.canoo.com/> (2004-05-02).
- Arie van Deursen, L. M. (2002), 'The video store revisited – thoughts on refactoring and testing'.
- Bach, J. (1996), 'Test automation snake oil', *Windows Tech Journal* .
- Beck, K. (2000), *Extreme Programming Explained*, Addison Wesley.
- Beck, K. (2001), 'Aim, fire!', *IEEE software* .
- Beck, K. (2002), *Test-Driven Development: by Example*, Addison Wesley.
- Caswell, M., Aravamudhan, V. & Wilson, K. (2004), 'Jfcunit', <http://jfcunit.sourceforge.net/> (2004-05-02).
- Crispin, L. (2002), *Testing Extreme Programming*, Addison-Wesley Pub Co.
- Cunningham, W. (2004), 'Fit homepage', <http://fit.c2.com/> (2004-05-02).
- Dave Thomas, A. H. (2002), 'Mock objects', *IEEE Software* .
- Fewster, M. & Graham, D. (1999), *Software Test Automation: Effective use of test execution tools*, Addison Wesley / ACM Press.
- Finsterwalder, M. (2001), Automating acceptance tests for gui applications in an extreme programming environment, in 'XP2001'.
- Gold, R. (2004), 'Httpunit homepage', <http://httpunit.sourceforge.net> (2004-05-02).
- Jeffries, R. (2001), 'What is extreme programming?', *XProgramming.com* .
- Jeffries, R. (2004), 'Xprogramming / software downloads', <http://www.xprogramming.com/software.htm> (2004-05-02).
- Kaner, C. (1997), 'Improving the maintainability of automated test suites'.
- Koss, R. (2002), 'Presentation: Testing things that seem hard to test', <http://www.objectmentor.com/resources/articles/TestingThingsThatAreHa~9740.ppt> (2004-05-02).
- Meszaros, G. (2003), 'Agile regression testing using record and playback', *XP/Agile Universe 2003* .
- Muckridge, R. & Tempero, E. (2004), 'Retrofitting an acceptance test framework for clarity'.
- Ron Jeffries, Ann Anderson, C. H. (2000), *Extreme Programming Installed*, 1st edition edn, Addison-Wesley Pub Co.
- Swan, A. (2002), Is gui testing difficult?, in 'Workshop on Testing in XP, WTiXP 2002'.