Development and Maintenance Efforts Testing Graphical User Interfaces: A Comparison

Antonia Kresse
Berner & Mattner Systemtechnik GmbH
Berlin, Germany
antonia.kresse@berner-mattner.com

Peter M. Kruse
Berner & Mattner Systemtechnik GmbH
Berlin, Germany
peter.kruse@berner-mattner.com

ABSTRACT

For testing of graphical user interfaces many tools exists. The aim of this work is a statement regarding the advantages and disadvantages of various testing tools with regard to their use in the economic context to be taken. It is compared, inter alia, whether there are differences in the generations of test tools in terms of finding defects and which tool has the lowest development and maintenance costs. Results show that with QF-Test test suites can be created the quickest while EggPlant has the shortest maintenance time. TestComplete performs worse in both disciplines. For test robustness, no clear picture can be drawn. The selection of a test tool is typically done once in a project at the beginning and should be considered carefully.

CCS Concepts

•**Software and its engineering** → *Software testing and debugging;* •**Human-centered computing** → User interface programming;

Keywords

GUI Testing; Empirical study

1. INTRODUCTION

In many industries driven by the market, the time between releases of new software products has significantly shortened. This results in a higher pressure and hence shorter development cycles for commercial manufacturers. The high customer expectations regarding quality are of major importance and give the software testing and debugging an important role [11]. Consequently, testers can no longer test manually, as this is too lengthy, costly and error-prone [7]. As part of the ever-changing requirements, the tests must be run on a regular basis to check the quality of the software continuously (regression testing) [23]. Automated tests can be a good solution [9].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

A-TEST'16, November 18, 2016, Seattle, WA, USA
ACM. 978-1-4503-4401-2/16/11...$15.00
http://dx.doi.org/10.1145/2994291.2994299

Most automated test techniques work on a deeper level, a well-known example is unit testing [22]. With the help of these tests defects in various software components of the system under test (SUT) can be found, however, in this way the interactive view of the user through the graphical user interface (GUI) will not be checked. In addition to the underlying functions of a system, the GUIs are not negligible. Their increasing importance is also reflected in the source code, in which the program code for the graphical user interface itself can take up to 60% [19]. Accordingly, it is important to consider the functionality of the GUI when testing the SUT. For the GUI tests, capture and replay tools are used which record mouse and keyboard events and play them back again. Testing with these tools simulate actions of end users and thus constitutes a good way of GUI testing [13, 19]. Furthermore, the tests can be played back as often as needed, which is why they are more effective than the manual testing of the GUI.

The development of tests with capture and replay tools, however, is very labor and time-consuming [19], which is why it is particularly important that the tests are created with a reasonable quality and not lead to errors with repeated executions. Even the smallest changes in the source code may lead to failing tests [19].

1.1 Background

Capture & Replay is a test method which is based on two steps [6]: In a first step mouse and keyboard events are recorded and automatically saved in a script, which can be replayed in the second step. To date, there are three generations of this testing technique, which are distinguished by their different interactions with the SUT. The first generation accesses the SUT using coordinates of the surface [8]. This variant is somewhat error prone and already leads to complications with the smallest changes in the GUI. Thus alone a higher resolution of the screen can cause to fail tests. Due to the high maintenance costs and lack of robustness, this generation is no longer used today [3, 13]. The second generation of GUI-based testing tools directly accesses the components / widgets of the GUI and thus interacts with the SUT. The tools use the properties of GUI components and extract for example the name, ID, type, etc. and interact with them. This makes the tools of the second generation more robust against minimal changes in the presentation, such as a change in resolution, which no longer have to lead to errors. However, the components can change in development or maturity, thus the identification of components fails which again leads to failing tests [21]. Furthermore, this ap-

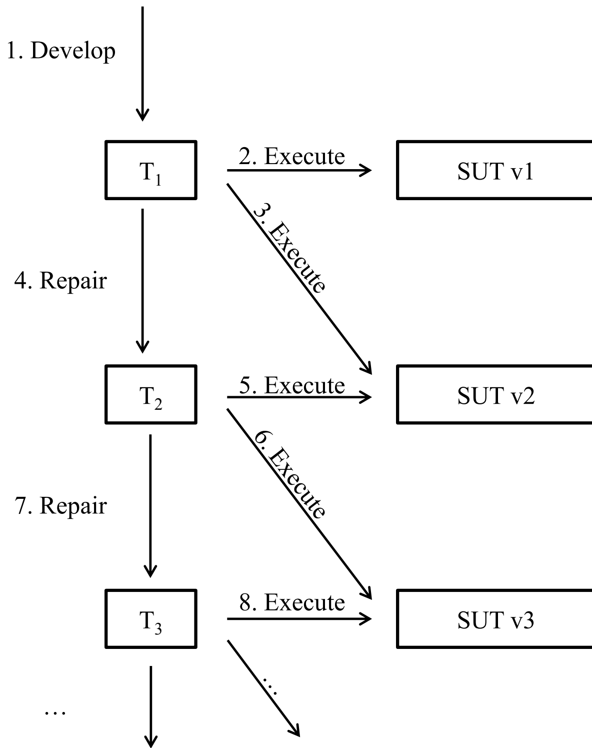

Figure 1: Study Design

proach does not correspond to the normal behavior of a user and does not reflect completely the clicks on the surface. A new method is the Visual GUI Testing (VGT), which interacts via image recognition with the SUT and thus forms the third generation [3]. The idea of VGT emerged already in the early 90s, however, has been only been realized now in the industry due to the required high and strong hardware resources. The main difference from Visual GUI Testing to the other variants, the ability of image recognition, so test scripts can be created that can interact with each graphical component on the monitor. These include buttons, text boxes, images, and more. Furthermore, the image recognition allows a way of testing which is the most similar to user behavior. It just interacts with shown elements in a flat structure, without using properties of the objects, etc.

2. STUDY DESIGN

We propose the following tool comparison: A set of test programs is selected and the same tests for a selected SUT are recorded with them. A conventional desktop application will be used as SUT. The tests are first designed for the first version of the SUT and then executed on the second and third version. The aim is to compare the necessary development and maintenance costs of testing tools with each other and so to find out which tools are best suited for economic purposes with respect to a constantly changing SUT. For the proposed study, only one SUT is used, which will be precisely described in the work with respect to its properties such as size, release cycles, etc. The complexity of the underlying source code is not in focus, as the tools deal with the GUI [3].

Table 1: TESTONA Releases

	4.1 to 4.3	4.3 to 4.5
Time between Releases	37 weeks	27 weeks
Tickets	22	111
- New Features	10	27
Changed LOC		
- Changed Files	187	488
- Added LOC	2814	8177
- Deleted LOC	1936	4504

The study design for each single tool is described in Figure 1. First tests T_1 for version 1 are developed (step 1). Subsequently, the tests are executed on version 1 (step 2) and then on version 2 (step 3) to check which tests are broken due to the changes between the two versions (1 and 2). In step 4, the tests T_1 will be repaired resulting in the second version of tests T_2, which are executed against version 2 (step 5) and version 3 (step 6) of the SUT. In step 7, the tests T_2 are adapted for SUT version 3 (resulting in T_3) and are consequently executed against SUT version 3 (step 8). This approach can be repeated for further versions of the SUT.

Several aspects are considered: First, when designing (step 1) and repairing the tests (steps 4 and 7) the time required for developing / repairing of tests is considered. In the execution steps (steps 2, 3, 5, 6, and 8) we first look at the number of failing tests. Subsequently we consider what kind of test are not running through (possibly they can be categorized) and why (tool or development problem).

The above described study design is executed by a set of different test tools and then provides a respective result per test tool. These results are finally compared among each other to draw pros and cons between the individual tools.

3. EXECUTION

The SUT is **TESTONA**,[1] a desktop application programmed in Java using the Eclipse platform running on the Windows operating system (Figure 2). TESTONA is the graphical editor [16, 27] for the classification tree method [10], a black box test design technique. Two main steps are performed to create a classification tree: **Identification** of relevant factors (*classifications*) with their values (*classes*) of a system under test and **Combination** of a single class from each classification into test cases.

The initial set of tests is developed for TESTONA 4.1. Maintenance efforts are then considered for TESTONA 4.3 and TESTONA 4.5. Changes between different versions of the SUT are given in Table 1.

Though the creation of TESTONA 4.3 took longer than TESTONA 4.5, it contains less new features. This is also reflected by the number of changes, both in terms of lines of code (LOC) and number of touched files.

Test scenarios are then chosen by analyzing tickets from change report (Table 1) with respect to affected GUI elements in the application. Therefore, results are not representative for a complete GUI test suite and its maintenance between releases.

26 test scenarios containing as many different parts of the software and underlying GUI components as possible are

[1]http://www.testona.net/

Table 2: Used GUI Components in Tests

	Buttons	Menu	Menubar	Textbox	Popup Menu	Tree	Toolbar	Dropdownmenu	Tree Node	TestcaseTree	Sub Node	Properties	Listview Entry	Checkbox	Radiobutton	Tabs / Tabbar	Treetable	Treeview	Tabelle	Slider	Color Picker	Link	Combobox	Scrollbar
CreateNewTree	1	2	2	1	4	9	4																	
EditTree	6	3	2	1	6	11	5					2								1	2			
FoldTree	1	2	2	1	5	8	1									1								
CopyPasteOfSubtrees	4	4	4	2	6	7	2																	
WriteProtectedFiles	4	5	5	2		4	1	1								1								
PropertiesView	5	4	4	3		2	3					6												
Autolayout	15	3	2	2		3						2								2				
Outline1	6	3	2	1	1	1												4						
Outline2	4	3	2	3	12		2											9						
CreateRenameTests	5	3	2	3	5	2				3		2					7							
ChangeMarktypes	6	4	4	1	3			2	1	1	2						10							
GenerateTestcases	16	6	6	3	2			1	2	4					2								3	
ManuellTestcases	12	4	4	4		2			2							1								
RenameGroups	5	2	2	5	5				4	5	7													
TestSequences	2	2	2	1	5				8		3													
BoundaryValues	10	4	4	8				1																
CreateEditTags	12	4	4	3		6	3						4	1										1
VariantsView	10	7	6	2	6	2	4	2	1		1		1			1					1			
Help	7	4	4	2																				
SearchingElements	6	3	3	5	7		2	2		1				5										
Logical Dep.	14	4	4	3			2												3					
Numeric Dep.	11	4	4	2		2	1	3		2					1		1		1					
ExportExel	10	5	5	3				2	3		2		2	1										
ExportHTML	7	4	4	4				1			1		2	1										
Import	7	2	2	2					1	1	1		1		2									
Testcoverage	1	4	4	1			2	1		1		1		1	1		1					4		
Clicks	117	95	89	68	67	59	32	16	22	18	17	13	10	9	6	4	19	13	4	3	3	4	3	1
In Tests	26	26	26	26	13	13	13	10	8	8	7	5	5	5	4	4	3	2	2	2	2	1	1	1

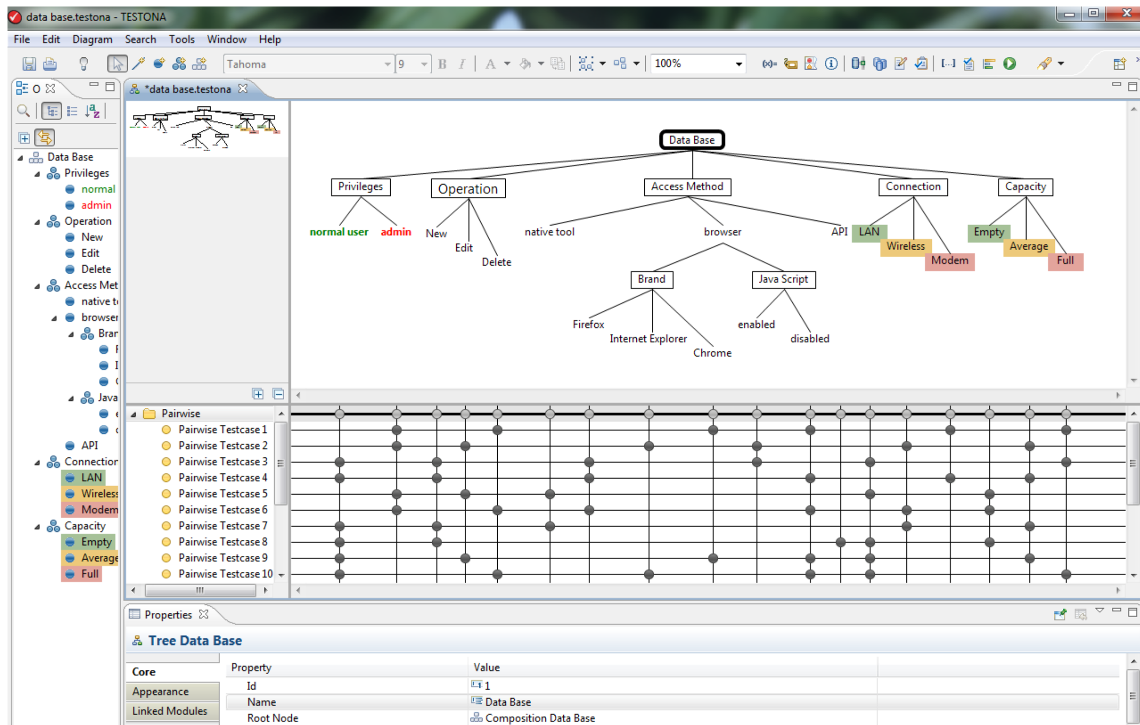

Figure 2: TESTONA Main Window

specified (Table 2). Lines contain the respective test scenario, columns indicate number of included GUI components (e.g. buttons, text boxes, check boxes, radio buttons, etc.). The last two lines indicate the total number of clicks on items per type and the occurrence of each component type in tests.

We have grouped the test scenarios according the main steps of the classification tree method. The first nine tests (`CreateNewTree`, ..., `Outline2`) cover the creation and layout of the classification tree. The `CopyPasteOfSubtrees` test contains copy and paste operations using the system clipboard. The `WriteProtectedFiles` test contains file handling, especially an attempt to save a write-protected file.

In the second group (`CreateRenameTests`, ..., `TestSequences`) we have the creation of test cases and test case handling in general, covering the seconds main step of the classification tree method.

The last group (`BoundaryValues`, ..., `Testcoverage`) contains enhancements to the classification tree method—such as constraints (`LogicalDep.`, `NumericalDep.`) [15] or the annotation of meta-data using tags (`CreateEditTags`) [18]—and features of TESTONA, that are not explicitly part of the classification tree method—such as `Import` and `Export` functions [27].

For this work, we use **Eggplant**,[2] **QF-Test**[3] and **TestComplete**.[4] Eggplant (Version 16.01) is considered a third generation Visual GUI Testing tool, QF-Test (Version 4.0.10) and TestComplete (Version 11.31) are second generation *conventional* Capture and Replay Tools (according to the classification by Alégroth et al. [2, 3]). Tools are chosen arbitrarily based on their availability and their perceived market share. All three of them are actively marketed by their respective vendors for the same task: Interactive GUI Testing. All vendors offer online training and webinars (2–3h each), which we have participated in.

The 26 specified tests are now implemented (in the same order as given in Table 2) for the first GUI testing tool: Eggplant. The test implementation is then executed against TESTONA 4.3. This allows for identification of broken tests requiring maintenance efforts. Maintenance is done in the same order as given Table 2. The step is then repeated, the now repaired tests are executed against TESTONA 4.5 to identify broken tests and perform test suite maintenance.

The whole procedure (implementation of 26 tests, running against 4.3, maintenance, running against 4.5, maintenance) is then repeated with both, QF-Test and TestComplete.

4. RESULTS

Results from the execution can be found in Table 3. The first column in the table gives the name of the test. The next columns give the details for the initial test creation with all three tools. All times are given in Minutes [m]. Both QF-Test and TestComplete have an additional column for initial maintenance. The second and third group of columns then give expected test breaking (indicated with ↯ in the table) and the actual results for maintenance effort.

4.1 Initial Implementation

For each tool, the implementation was done in table order. For each tool, early test scenarios needed noticeably more time than later tests (Table 3).

Initial implementation of 26 test scenarios took 2750 minutes (45h 50m) with Eggplant. An average test case needed 105m 20s (1h 45m 20s) for implementation. The `Help` test was implemented the fastest with 30 minutes, the `ChangeMarktypes` test with 240 minutes (4h) took the longest.

For QF-Test, the initial implementation required 1395 minutes (23h 15m). Additional 85 minutes (1h 15m) were needed to have test cases stably executing on the test execution environment. An average test case needed 56m 24s for implementation (and stabilization). The `TestSequences` and `Testcoverage` tests were implemented the fastest with 15 minutes, the `Help` test with 335 minutes (5h 35h) took the longest.

In TestComplete, the initial implementation was finished in 1735 minutes (28h 55m). Additional 915 minutes (15h 15m) were required for test stability on the execution environment. An average test case needed 101m 24s (1h 41m 24s) for implementation (and stabilization). The `Logical Dep.` test was implemented the fastest with 15 minutes, the `PropertiesView` test with 240 minutes (4h) took the longest.

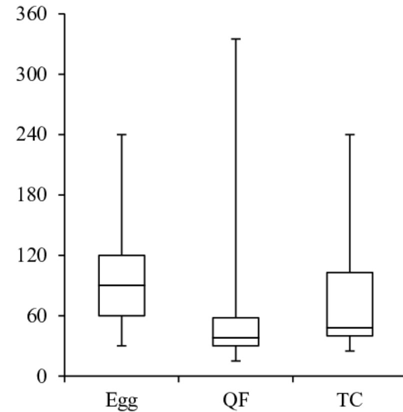

Figure 3: Implementation (Time in Minutes [m])

Conclusion: For the initial implementation, the tools perform quite differently (Figure 3).

QF-Test was the fastest with a total time needed of 24h 30m, or 56m 24s average per test case. It requires some stabilization effort, but was still almost twice as fast as its competitors.

TestComplete was the second fastest with a total time of 44h 10m, or 101m 24s (1h 41m 24s) average per test case. It requires a massive stabilization effort.

Eggplant took the longest with 45h 50m, or 105m 20s (1h 45m 20s) average per test case. It, however, does not require any stabilization effort.

4.2 Robustness

Based on the change report and our ticket analysis (Table 1), we expected three test scenarios (`SearchingElements`, `ExportHTML`, `Import`) to require maintenance activity (Table 3, indicated with ↯) from TESTONA 4.1 to TESTONA 4.3.

For Eggplant, two of the expected tests (`SearchingElements`, `ExportHTML`) where actually broken due to changes to the GUI. One test case (`Import`) was still working though breaking was expected, because all required GUI elements

[2]http://www.testplant.com/eggplant/
[3]http://www.qfs.de/en/qftest/
[4]http://smartbear.com/testcomplete/

Table 3: Implementation and Maintenance Efforts (Time in Minutes [m])

	4.1					4.3				4.5			
	Egg	QF		TC			Egg	QF	TC		Egg	QF	TC
CreateNewTree	180	60		150	+80							20	
EditTree	180	90		130	+30						10		
FoldTree	120	60		240								5	
CopyPasteOfSubtrees	120	45		20	+195								
WriteProtectedFiles	60	30		30									
PropertiesView	120	30		240									
Autolayout	150	30		70									
Outline1	60	30		70	+20								
Outline2	90	60		40									
CreateRenameTests	120	20	+20	45									
ChangeMarktypes	240	30		40						ϟ	60	10	25
GenerateTestcases	120	30		40						ϟ	25	45	40
ManuellTestcases	120	105		45									
RenameGroups	120	60		40						ϟ	5	25	15
TestSequences	90	15		40							10		
BoundaryValues	60	20		30	+40								
CreateEditTags	90	40		50				20					
VariantsView	90	45		25								5	80
Help	30	320	+15	30	+100						5	5	
SearchingElements	120	40	+10	30	+130	ϟ	5	20	10	ϟ	20	20	40
Logical Dep.	60	20		15						ϟ	20	10	60
Numeric Dep.	60	50		35						ϟ	10	20	30
ExportExel	60	15	+20	20	+90		10		15	ϟ	5	10	10
ExportHTML	60	20		20		ϟ	15	10	20	ϟ		35	25
Import	60	15	+20	30	+130	ϟ			20		10	15	45
Testcoverage	60	15		15	+30								
Auxiliary	110	100		195	+70								
TOTAL	2750	1395	+85	1735	+915		30	50	65		180	225	370

were still present in recognizable form. An additional test (`ExportExcel`) broke, which was not anticipated, because of GUI changes in dialogs not reflected by tickets in the change report.

QF-Test had similar results as Eggplant: The same two tests (`SearchingElements`, `ExportHTML`) required changes due to changes in the GUI. Again, the (`Import`) test was working unexpectedly. There also was an additional test case breaking, which was `CreateEditTags` in this case: The internal structure of some dialog button had changed.

TestComplete performed slightly worse: All three expected tests broke. Additionally, also the `ExportExcel` test broke (similar to Eggplant), giving a total of four test cases to be maintained.

From TESTONA 4.3 to TESTONA 4.5, again based on change report and ticket analysis, we expected eight test scenarios to require maintenance (Table 3, indicated with ϟ). In Eggplant, there were 11 tests requiring updates due to GUI changes, containing seven of the eight anticipated tests. One test case (`ExportHTML`) was still working unexpectedly. Although there were several added GUI elements, the test was still working. There were four unexpected broken tests (`EditTree`, `TestSequences`, `Help`, `Import`). For the first two, it was due to changed elements in popup menus (also reflected by tickets, but missed in our analysis). The `Help` test failed due to then longer loading times of the help window. The `Import` test was affected by changes to the internal structure of listview entries.

In the QF-Test test suite, there were 13 broken tests, containing all eight expected ones. The five additional broken tests were (`CreateNewTree`, `FoldTree`, `VariantsView`, `Help`, `Import`). For the first two, it was again due to changed elements in popup menus. The `VariantsView` test failed due structural changes to the GUI (all elements were still their, having slightly different containers). Reasons for `Help` and `Import` tests failing are similar to Eggplant.

For TestComplete, there were 10 broken test cases, containing all eight expected ones. The two additional broken tests were (`VariantsView`, `Import`), caused by the same reasons as with QF-Test.

Conclusion: In terms of robustness, the tools perform quite similar. Eggplant has a total of 14 tests requiring maintenance. TestComplete also has also 14 tests. QF-Test performs slightly worse with 16 broken tests between different releases of the SUT.

There is not significant different between generations of capture and replay tools in terms of robustness.

4.3 Maintenance Effort

For Eggplant, maintenance took 30 minutes total (10 minutes average for 3 tests) from TESTONA 4.1 to TESTONA 4.3. From TESTONA 4.3 to TESTONA 4.5 maintenance took 180 minutes (3h), that is 16m 22s average for 11 tests. Maintenance effort from both releases was 210 minutes (3h 30m), averaging to an effort of 15m for each of 14 broken test cases.

For QF-Test, maintenance took 50 minutes total from TESTONA 4.1 to TESTONA 4.3 (16m 40s average for 3 tests). From TESTONA 4.3 to TESTONA 4.5 maintenance was done in 225 minutes (3h 45m) for all 13 broken tests (average maintenance time of 17m 18s). For both releases combined, maintenance time was 275 minutes (4h 35m) including 16 broken tests, with an average of 17m 11s.

With TestComplete, maintenance from TESTONA 4.1 to TESTONA 4.3 was done in 65 minutes (1h 5m) for 4 broken tests (average of 16m 15s). From TESTONA 4.3 to TESTONA 4.5, maintenance for 10 broken test cases re-

quired 370 minutes (6h 10m), averaging to 37m per test. The combined maintenance effort was 435 minutes (7h 15m), the effort for fixing all 14 broken tests was 31m 4s.

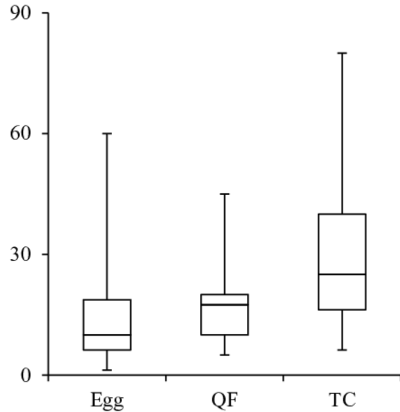

Figure 4: Maintenance (Time in Minutes [m])

Conclusion: The maintenance effort was quite different between the tools (Figure 4). Eggplant performs best with the shortest overall fixing time (3h 30m) and the shortest average time per test (15m).

QF-Test scores second, the overall fixing time is higher (4h 35m), the average time per test is longer (17m 11s), but still close to Eggplant.

TestComplete's performance is the worst of the three contenders: Overall fixing time is 7h 15m, time needed to fix an average test case is 31m 4s, both values doubling Eggplant.

4.4 Discussion

From the detailed results, there are three main insights:

- Initial Implementation is fastest with QF-Test, both Eggplant and TestComplete take much longer.
- Robustness offers no clear picture, all tools perform quite similar.
- Maintenance Effort is best with Eggplant, with QF-Test still ok. TestComplete shows weak performance.

From our figures, QF-Tests offers the best initial performance. In our case, EggPlants maintenance advantage over QF-Test would pay off after 581 changes. Due to its performance, we would not recommend TestComplete for future GUI testing tasks.

The rather slow pay off of EggPlant is in line with latest results by Alégroth et al.[1].

4.5 Threats to Validity

This section discusses some of the threats addressed in [24].

Construct validity. With respect to the efficiency and learning effort, we could not fully mitigate the threat caused by self-reported working times (e.g. by means of working diaries). Accuracy of these measures could have been affected by other (e.g. social psychological) factors. However, using other complementary measures (i.e. SVN logs) helps to triangulate the observations.

Internal validity. The quality of the test implementations could have been affected by the level of testing experience. Although a training program was implemented this threat could be only reduced.

External validity. The obtained results about the applicability of GUI tools need to be evaluated with more SUTs. Another issue is the very limited number of participants in this study. Regarding the GUI testing tools, they were carefully selected by the testers.

5. RELATED WORK

There are not many comparisons on the economic aspects of GUI test creation and maintenance. A Master's Thesis by Khazal and Sigurdsson does a comparison with just two tools involved [14]. They compare HP QuickTest Professional against HP Unified Functional Testing. 20 test scenarios are implemented in both tools. Their focus also is on development and maintenance time and robustness. Unfortunately, details on the SUT are not disclosed.

Alégroth et al. compare manual system test cases with their Visual GUI Testing approach [1, 2].

A large body of work is focuses on techniques of capture and replay and their advances. In our work we rely on conventional GUI testing using capture and replay. Several authors suggest that using model-based GUI testing might have advantages, especially when models can be used for the generation of actual GUI tests [5, 20, 25].

Bauersfeld and Vos implement a tool for testing GUI system: GUITest [5]. Their tool provides the following features: *a)* Works on all native GUIs, which are recognized by the Windows API. *b)* SUT must not be instrumented. *c)* Allows the user to define their own actions. *d)* Generated Test sequences can be stored and played back.

Memon et al. also offer an implementation for model-based GUI testing, with prototypical capture, semi-automatic modeling and automated execution [20]. Memon et al. model the SUT with GUI forests, event-flow graphs and integration trees.

Stadie and Kruse [25] have a similar approach, however, they rely on state charts [12] and classification trees [10]. They propose that state charts are more suitable, because they are more compact due to hierarchies and orthogonality.

In the context of web applications, there are similar approaches [4, 17].

6. CONCLUSION

In this work we perform a comparison of GUI testing tools in terms of creation and maintenance efforts, laying a strong focus on the economic aspect of testing. We see differences with respect to different tool generations and in their capability of finding defects. The search for the tool with the lowest development and maintenance cost is a main motivation for carrying out the comparison. With respect to that question, we recommend QF-Test for fast creation of test suites, though EggPlant has a small advantage when it comes to maintenance efforts in the long run.

For future work we see the in-depth analysis of the data acquired, e.g. individual causes for failing test scenarios. Also the impact of used GUI elements (Table 2) on implementation and maintenance effort as well as robustness will be further analyzed. This might allow for gaining development recommendations for low-maintenance test modeling.

We also plan to compare with different approaches (such as TESTAR [26]) for the very same SUT versions. Also we need to investigate further in bug finding capabilities.

7. REFERENCES

[1] E. Alégroth, R. Feldt, and P. Kolström. Maintenance of automated test suites in industry: An empirical study on visual gui testing. *Information and Software Technology*, 73:66–80, 2016.

[2] E. Alégroth, R. Feldt, and L. Ryrholm. Visual gui testing in practice: challenges, problemsand limitations. *Empirical Software Engineering*, 20(3):694–744, 2015.

[3] E. Alégroth, Z. Gao, R. Oliveira, and A. Memon. Conceptualization and evaluation of component-based testing unified with visual gui testing: an empirical study. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–10. IEEE, 2015.

[4] D. Amalfitano, A. R. Fasolino, and P. Tramontana. Rich internet application testing using execution trace data. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 274–283. IEEE, 2010.

[5] S. Bauersfeld and T. E. Vos. Guitest: a java library for fully automated gui robustness testing. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 330–333. ACM, 2012.

[6] E. Börjesson and R. Feldt. Automated system testing using visual gui testing tools: A comparative study in industry. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 350–359. IEEE, 2012.

[7] M. Finsterwalder. Automating acceptance tests for gui applications in an extreme programming environment. In *Proceedings of the 2nd International Conference on eXtreme Programming and Flexible Processes in Software Engineering*, pages 114–117, 2001.

[8] M. Grechanik, Q. Xie, and C. Fu. Creating gui testing tools using accessibility technologies. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, pages 243–250. IEEE, 2009.

[9] M. Grechanik, Q. Xie, and C. Fu. Experimental assessment of manual versus tool-based maintenance of gui-directed test scripts. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 9–18. IEEE, 2009.

[10] M. Grochtmann and K. Grimm. *Software Testing, Verification and Reliability*, volume 3. John Wiley and Sons, second edition, July 1993.

[11] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[12] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[13] E. Horowitz and Z. Singhera. Graphical user interface testing. *Technical eport Us C-C S-93-5*, 4(8), 1993.

[14] A. Khazal and A. D. Sigurdsson. Component-based capture & replay vs. visual gui testing: an empirical comparison in industry. Master's thesis, Göteborg: Chalmers tekniska högskola, 2015.

[15] P. M. Kruse, J. Bauer, and J. Wegener. Numerical constraints for combinatorial interaction testing. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 758–763. IEEE, 2012.

[16] P. M. Kruse and M. Luniak. Automated test case generation using classification trees. *Software Quality Professional*, 13(1):4–12, 2010.

[17] P. M. Kruse, J. Nasarek, and N. C. Fernandez. Systematic testing of web applications with the classification tree method. In *Proceedings of the XVII Iberoamerican Conference on Software Engineering (CIbSE 2014)*, 2014.

[18] E. Lehmann and J. Wegener. Test case design by means of the CTE XL. *Proceedings of the 8th European International Conference on Software Testing, Analysis and Review (EuroSTAR 2000), Kopenhagen, Denmark, December*, 2000.

[19] A. M. Memon. Gui testing: Pitfalls and process. *Computer*, 35(8):87–88, 2002.

[20] A. M. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 260–260. IEEE Computer Society, 2003.

[21] R. Miller and C. T. Collins. Acceptance testing. *Proc. XPUniverse*, 2001.

[22] M. Olan. Unit testing: test early, test often. *Journal of Computing Sciences in Colleges*, 19(2):319–328, 2003.

[23] A. K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Communications of the ACM*, 41(5):81–86, 1998.

[24] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131–164, 2009.

[25] O. Stadie and P. M. Kruse. Closing gaps between capture and replay: Model-based gui testing. In *Proceedings of 1st INTUITEST Workshop*, 2015.

[26] T. E. Vos, P. M. Kruse, N. Condori-Fernández, S. Bauersfeld, and J. Wegener. TESTAR: Tool support for test automation at the user interface level. *International Journal of Information System Modeling and Design (IJISMD)*, 6(3):46–83, 2015.

[27] U. Zeppetzauer and P. M. Kruse. Automating test case design within the classification tree editor. In *Federated Conference on Computer Science and Information Systems 2014 (FedCSIS), 5th International Workshop Automating Test Case Design, Selection and Evaluation (ATSE)*, 2014.